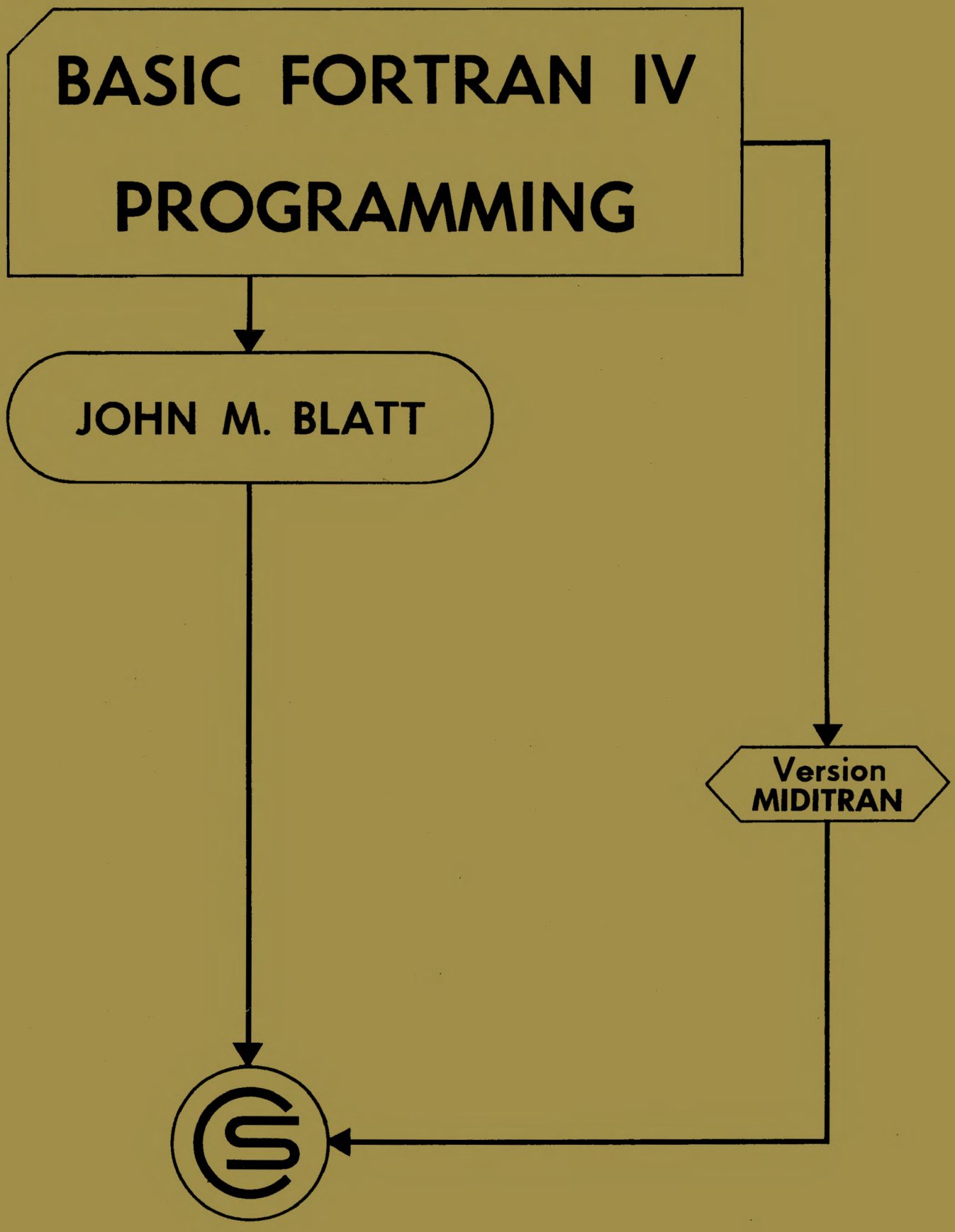
BASIC FORTRAN IV PROGRAMMING

JOHN M. BLATT



BASIC FORTRAN IV PROGRAMMING

BASIC FORTRAN IV PROGRAMMING

JOHN M. BLATT

PROFESSOR OF APPLIED MATHEMATICS
UNIVERSITY OF NEW SOUTH WALES
SYDNEY, AUSTRALIA

COMPUTER SYSTEMS (AUST.) PTY. LTD., SYDNEY, AUSTRALIA.

Printed in Australia by Simmons Limited, Sydney

National Library of Australia registry number Aus 69-1970.

Registered in Australia for transmission by post as a book

PREFACE

This book provides a rapid introduction to computer programming in FORTRAN IV, for people with no computer experience and no mathematical training beyond elementary High School mathematics. The emphasis is on practice at programming and debugging, with problems taken from both data-processing and scientific applications.

The language taught is *USA Standard Basic FORTRAN IV.* This language is available on *every* computer, and suffices for nearly all practical applications. Thus, *after this course, the student can program any computer, not merely the particular computer on which he has been taught.* More elaborate versions of FORTRAN are unsuitable for our purpose, because: (1) They contain too much material for a rapid programming course; and (2) Many students are taught on small computers which accept only Basic FORTRAN IV.

Computer programming is a practical skill which requires actual experience. No amount of theoretical reading will enable you to drive a car; precisely the same is true of programming a computer. This book has been designed so that you can prepare programs and run them on the computer, *right from Chapter I on.* There are two kinds of exercises:

1. *Drill exercises,* which can be done with pencil and paper, and for which the answers are given in this book, starting on page 153; and
2. *Programming exercises,* which involve writing a complete computer program and running this program on the machine. These exercises are considered satisfactory only after the program has run to completion, without producing error messages, and giving numerically correct results. The answers to the programming exercises are available to teachers only, in a separate booklet.

The remainder of this preface is addressed to teachers, not to students, and explains how this book has overcome certain technical problems of teaching FORTRAN.

Until Chapter VII, "cookbook" FORMAT statements are used to print out numbers, print out lines with messages, and to read input data cards. In Chapter VII, the FORMAT statement is explained in detail. We use symbolic names (NREAD and NPRNT) for the unit numbers of the card reader and line printer, respectively. This makes all programs in this book acceptable at all installations, subject at most to a trivial change of the first two statements ("NREAD=1" and "NPRNT=3"), so as to accord with local useage. Students are told to enquire about these unit numbers, and to alter these two statements if necessary.

The text is organized for easy learning in stages, not as a manual. But concise summaries appear at the end of every chapter, and there is also a condensed manual, the "Glossary of Basic FORTRAN IV", on pages 150, 151, and 152. These three pages contain everything needed for rapid reference in practical programming: character set, card layout, sequence of statements in a program unit, list of all statement types with illustrative examples, list of built-in functions, etc.

The FORTRAN language by itself is unfortunately not enough. Before the student can do anything at all on the computer, he must be told how to prepare input cards, how to make up a job deck, and how to interpret the printed output.

In most FORTRAN books, this material is either absent (thus forcing the use of mimeographed handouts, or of manufacturers' manuals), or it is restricted to one particular computer and one particular version of FORTRAN.

In this book, we have developed an entirely novel method of overcoming this difficulty. The book consists of a "constant" part and a "variable" part. The "constant" part includes Chapters I to XIII, inclusive, the FORTRAN Glossary, and the solutions to all the drill exercises. This occupies the first 160 pages.

The "variable" part, Part C, is quite short. It starts on page 161. These concluding pages contain four more Chapters, each of these Chapters being available in several different "versions", to suit the particular School or College. For example, Chapter XIV tells the student how to prepare input cards. There is one version of this Chapter for the IBM 29 keypunch, and another version for preparation of optically marked (rather than punched) cards. Still further versions will be prepared if there is sufficient demand.

The last three Chapters explain some particular version of FORTRAN and its associated monitor system. For example, there is one version of the last three Chapters for the IBM 1130, and another version for the IBM 360 "E-level FORTRAN" under the Disk Operating System (DOS).

In all cases, these last three Chapters deal, respectively, with the input card deck, with interpretation of the printed output from the computer, and with the peculiarities of the version of FORTRAN available on that computer.

This kind of material can be found in manufacturers' manuals, of course, but students (and often not just students) find these manuals extremely tough going. In this book, we restrict ourselves to what the student really needs to know, but that is explained carefully and in detail. For example, we do not tell students how to produce messages on the operator's console, or how to reserve space on the system disk for their private data sets. But we not only list all diagnostic messages and execution time error messages, but give detailed hints for many of them, to help the student determine what these messages mean for his particular program. This makes it unnecessary for a student to run to a tutor every time he gets a diagnostic message. With large classes, this is extremely important.

Thus, a School or College which adopts this book does not need to prepare special mimeographed handouts to students, nor need it rely upon manuals provided by the manufacturer. *Everything that is needed for practical teaching is right in the book.*

But at the same time, the book is not restricted to one particular computer or to one particular version of FORTRAN. The bulk of this book and the course based on it, including all the exercises and their solutions, is suitable for *every* computer and for *every* FORTRAN compiler. When the computer is replaced by a new model, or a new version of FORTRAN becomes available, all that the School needs to do is to ask the bookshop to order the same book, but with a different "version" of the "variable" part.

In conclusion, it is my pleasant duty to thank Miss Ruth Blatt, Mrs. A. Nikov, Dr. R. Storer, and Dr. G. Sved, for reading the manuscript in whole or in part, and for making most valuable suggestions for improvements; and to thank Mrs. Paula Payne and Mrs. Ingrid Kole for preparing the camera copy for the printer.

John M. Blatt
Sydney, Australia
June 1969

TABLE OF CONTENTS

PART A

This first part of the book gives just enough of the Basic FORTRAN IV language to enable you to write simple programs and put them on the computer. Most of the necessary information is the same for every version of FORTRAN. But a few things are different from one computer installation to another; these things are presented at the end of the book, in a form suitable for your particular computer and your particular version of FORTRAN. You will be referred to the appropriate pieces of information at the points where you first need them.

Practical experience in writing programs, preparing them for running on the computer, and correcting the inevitable programming mistakes, is indispensable in learning programming. Lectures, study sessions, private study all have their place; but there is no substitute for practical experience on the computer.

It is neither necessary nor desirable for you to read the whole book before putting your first program on the computer. On the contrary, the first "programming exercises" appear at the end of Chapter I, already, and should be attempted at that point. In addition, there are "drill exercises" at the end of every Chapter. Answers to the drill exercises are given starting on page 153. Answers to the programming exercises are available in a little booklet for teachers only.

No attempt is made in these early Chapters to make everything complete; details will be filled in later, in Part B. Rather, we have tried to make the material easy to follow for someone who has not had anything to do with computers before.

CHAPTER I

INTRODUCTION

Section A: Computers and Compilers.

An electronic computer is a machine that does calculations on data (in the form of numbers given to it) by obeying a sequence of instructions that also must be given to it. The machine does not "think", but merely obeys the given instructions without either will or common sense of its own.

The computer can add, subtract, multiply, and divide; it can "read" data from cards (see page 9 for the appearance of a card) inserted into a "card reader", and it can print out results. In order to solve any problem on a computer, the problem must first be analyzed into these elementary steps. Detailed instructions must then be given to the computer in a "language", called "machine language", which this particular computer has been constructed to read and obey. This set of instructions is called a "program", and the process of preparing the instructions is called "programming". In the form just described, it is called "machine language programming".

Machine language programming is difficult to learn, time-consuming, and furthermore a different "language" must be learned for each new model of computer. In order to make programming simpler, and largely independent of the particular computer, programming is usually done in some "high-level language", of which FORTRAN is the most prominent example. When the program is written in FORTRAN, the computer itself is directed to translate from FORTRAN into its own machine language. The translation process is called "compilation", and the very elaborate and complicated machine language program which instructs the computer to carry out this translation is called a "compiler". Every modern computer comes equipped with its own FORTRAN compiler, usually provided by the computer manufacturer.

When the program is written in FORTRAN, the process of getting results out of the machine consists of two separate steps, or "phases", namely the "compile phase" and the "execution phase".

During the "compile phase", the machine is under control of the instructions in the compiler. Obeying these instructions, the machine reads the FORTRAN language program from FORTRAN language input cards prepared by the programmer. Having read each card, the machine translates the information on that card from FORTRAN into machine language. Note that your FORTRAN program does nothing at all during the compile phase; rather, something is being done to it: it is being translated from one language to another.

Some other, helpful operations are carried out at the same time. As each card of the FORTRAN program is read, its content is reproduced ("listed") in one line on the line printer (a typewriter types one character at a time, whereas a "line printer" prints one whole line at a time). The programmer can thus see, by looking at the printed

output from the machine, what the machine actually read from the card. Furthermore, the content of each card is checked to see whether it is grammatically correct FORTRAN. If a violation of the rules of the FORTRAN language is discovered, an error message ("diagnostic message") is printed out, to tell the programmer which rules of FORTRAN grammar have been broken. Grammatically wrong FORTRAN statements are not translated into machine language.

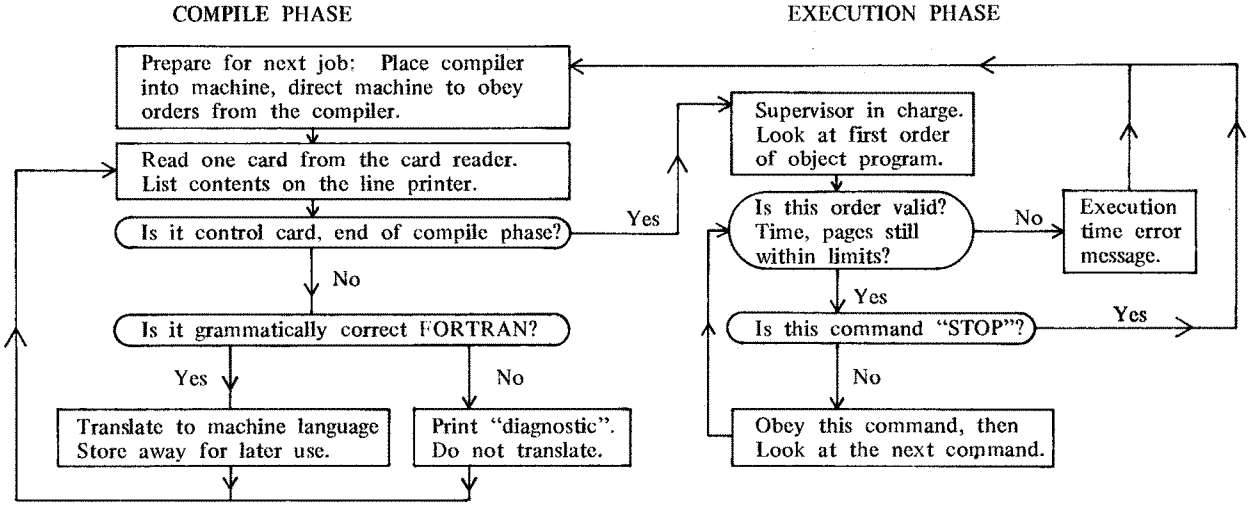
The net effect of the compile phase is the production of a translated program, called the "object program", which is in that computer's own machine language.

A special card, called a "control card", must appear at the end of the FORTRAN language input cards. This card serves as a signal to the compiler program that its job is finished. No more translation is required.

At this point, the "execution phase" starts. The machine is now directed to obey the first machine language order of the object program. Only at this moment does the machine begin to do what you told it to do when you wrote the FORTRAN program. Although no more translating is to be done, another master program called the "supervisor" or "monitor" now carries out a different function. As each command of the object program is obeyed by the machine, a check is carried out for attempts to do impossible things (like dividing by zero). Whenever such an attempt is detected, the supervisor takes over, prints out an "execution time error message" informing the programmer of what went wrong, and stops further program execution, usually. The supervisor also checks on the time taken during execution of the object program, and on the number of printed pages produced. Typical student programs should not require more than a few seconds of execution time, nor produce more than a few pages of printed output. The supervisor stops execution if these limits are exceeded.

After program execution has terminated (either because the machine finished all the commands in your program, or because execution was stopped by the supervisor), the supervisor calls for the input cards of the next student "job". This continues until all the "jobs" in the "job batch" are finished. (Not all computers operate precisely in the way we have just described, but the differences are not important to students.)

We now present a "flow diagram" to illustrate the compile phase and the execution phase. The little boxes contain information concerning the action to be performed, and the lines with arrows indicate what is to be done next. Some boxes contain "yes-or-no" questions; the line labelled "yes" is followed if the answer to the question is "yes"; otherwise, the line labelled "no" is followed.



Section B: FORTRAN Characters and Identifiers.

Most of this book is concerned with the FORTRAN language, and with the preparation of the source program in that language. From time to time, however, we shall need to consider what happens during translation (compile phase) and during execution of the object program.

The FORTRAN language is written with "characters" of three types:

1. *Alphabetic characters,* i.e., the ordinary letters A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P, Q,R,S,T,U,V,W,X,Y,Z. Only capital letters are used.
2. *Numeric characters,* i.e., the ordinary decimal digits (numerals) 0,1,2,3,4,5,6,7,8,9. The zero "0" and the capital Oh "O" are different characters, and so are the one "1" and the capital letter "I". These characters are easy to tell apart in a printed text, or in computer printed output. Some special care is needed in a hand-written program, however.(*)
3. *Special characters:* the period (or decimal point) ".", the comma ",", the asterisk "*", the equal sign "=", the plus sign "+", the minus sign "-", the slash (or division sign) "/", the left parenthesis "(", the right parenthesis ")", and, last but by no means least, the blank character " ". (**)

The term *alphameric character* is used for a character which may be either alphabetic or numeric (but not special).

Electronic computers are able to store, or "remember", numbers and instructions. The computer memory consists of a number of separate *memory locations,* each of which may be used to store a number. You may think of a memory location as one pidgeon hole, which can hold exactly one number at a time. The computer can "fetch" a number from some memory location, without affecting what is in the memory location. But if a number is "stored" into a memory location, then the number previously stored there is destroyed ("forgotten"). It is important to distinguish between the *memory location* and the *content of this memory location.* The same memory location may contain different numbers at different times, during program execution.

Memory locations are given "names" in FORTRAN programming. These names are called *identifiers.* For example, the identifier HEIGH is the name of a memory location in the machine. The memory location named HEIGH may contain the number 5.89 at one time, the number 7.39 at another time, and so on. The rule for constructing acceptable identifier names in Basic FORTRAN IV is:

(*) Some people use a slash through the zero "Ø" to distinguish it from capital "O". Unfortunately, some other people use the exact opposite convention, i.e., their hand-written symbol Ø means a capital "O". You should find out, and follow, the convention used at your own computer installation.

(**) Blank characters can be inserted, or removed, at will in most FORTRAN statements. But in certain situations (character strings within FORMAT statements, see Chapter IV, page 40) blank characters are counted as much as any other characters. Also, blank characters sometimes serve to separate two different things from each other. It is advisable to avoid blanks within one number, or within one identifier name. Within a number, blanks are usually taken to be zeros, which may or may not be what you want. If you do want a zero there, write it explicitly! Some FORTRAN compilers do not permit blanks within an identifier name.

In addition to the special characters listed here for Basic FORTRAN IV, full FORTRAN IV also permits the special character "$". (Perhaps because of the affection which IBM feels for this particular symbol, the character "$" is counted as an alphabetic (not special) character in some versions of IBM FORTRAN.)

Still other special characters are permitted by some dialects of FORTRAN, or in certain particular situations. We advise against use of any characters not in the standard character set.

An *identifier* consists of from *one to five alphameric characters, the first of which must be alphabetic.* These characters should appear next to each other, without intervening blanks. (*)

Examples: Some grammatically correct identifier names are: H, HA, HAHA, HEIGH, AREA, AREA1, A, AR, B8O0T, TOLER, RATIO, WIDTH. The character string 5AW is *not* an identifier, since the first character is numeric, not alphabetic. The character string VELOCITY is *not* an identifier, because it contains more than five characters. In practice, the restriction to no more than five characters is the most likely source of error in inventing identifier names.

The identifier name serves two separate and distinct functions: (1) It names the memory location where the information is stored; and (2) it tells what *type* of information is stored there. For the time being, we shall be interested in only one type of information: ordinary decimal numbers. These are called "real numbers" in FORTRAN jargon. (**) The FORTRAN rule is:

A *real variable identifier* is an identifier name starting with any letter other than I, J, K, L, M, or N.

Note: These six starting letters are reserved for integer variable identifiers; see chapters II and VI.

Examples: FOOL, DOPE, and STUPE are valid real variable identifiers. IDIOT, JOE, MORON, and KRAZY have incorrect starting letters for real variable identifiers.

Section C: A Simple FORTRAN Program

Rather than giving the full rules of the FORTRAN language first, and showing a FORTRAN program only afterwards, we shall now present an actual FORTRAN program in full, to show what it looks like.

The problem is: *A man owes a debt of $300.00, which he pays off with payments of $22.40 at the end of each month. An interest charge of 0.8 percent is made on the unpaid balance in each month. Compute the interest charge, the amount by which the debt is reduced each month, and the value of the debt that still remains to be paid. Print out all relevant information.*

Simple as this problem is, we shall have to wait till Chapter III to solve it in full. At the moment, we shall compute the numbers for only the *first* payment.

Let us compute the numbers "by hand" first, so that we know what to expect.

The interest charge for the first month is 0.8 percent of $300.00, that is, $2.40. This interest charge is subtracted from the man's payment, $22.40. This leaves $20.00 as the amount by which the debt is reduced as a result of the payment.

To get the value of the remaining debt, we subtract the debt reduction, $20.00, from the initial debt, $300.00. This leaves $280.00 as the remaining debt.

(*) This last rule is not enforced in USA Standard Basic FORTRAN IV, that is, the identifier name BOB could also be written as B O B, for example. However, a number of FORTRAN compilers do insist that there must not be any interior blanks inside identifier names.

(**) Since real numbers in any computing machine are carried to only a limited number of digits, a mathematically more correct term would be "rational numbers". Mathematicians distinguish between "real numbers" and "rational numbers", but we shall not do so in this book.

We present the FORTRAN program here in two different forms:

1. Printed onto a page, and
2. Handwritten onto a FORTRAN "coding sheet".

A third form, namely input cards to the computer, will be shown later.

The program consists of nine separate "FORTRAN statements", each of which is written on a separate line (and will later be transferred to a separate card):

```
1 FORMAT (7E16.6)
  DEBT=300.00
  PAYMT=22.40
  CHARG=0.008*DEBT
  REDUC=PAYMT-CHARG
  RMAIN=DEBT-REDUC
  WRITE (3,1) DEBT,CHARG,PAYMT,REDUC,RMAIN
  STOP
  END
```

Discussion and Explanation of this Program:

The *first statement* is required, but will not be explained until Chapter VII. All FORTRAN statements, including this one, are written starting in the *seventh* position on the line (they appear on cards *starting in column seven of the card).* The symbol "1" appearing in position 5 is *not* part of the FORTRAN statement; rather, it is a *statement number* assigned to this statement.

The *second statement* tells the compiler to produce an object program which takes the number 300.00 and stores it into memory location DEBT. Note that the identifier name DEBT starts with a letter other than I, J, K, L, M, or N. The number 300.00 is written as an ordinary decimal number *with a decimal point in it,* that is, as 300.00 rather than just 300 by itself. This is *necessary* for the time being, for all data numbers appearing in the program.

The *third statement* sets the value of memory location PAYMT. Note that we have shortened the English word "payment" to five-letter length, by omitting two of the letters.

The *fourth statement* computes the interest charge and stores its value into memory location CHARG. The interest rate of 0.8 percent has been converted to its fractional form, 0.008. The asterisk "*" is used to denote multiplication. *This asterisk must not be omitted.* For instance, the statement:

```
CHARG=0.008DEBT
```

would be ungrammatical FORTRAN. It would be diagnosed as an error by the FORTRAN compiler, and would not be translated into machine language at all. With the asterisk in its proper place, however, the statement is translated into the following sequence of commands: (i) fetch the number stored in memory location DEBT; (ii) multiply that number by 0.008; and (iii) store the product into memory location CHARG. Note that a single FORTRAN command is translated into several machine language commands.

The *fifth statement* computes the debt reduction REDUC, as the difference between the value of PAYMT (which has been set in the third statement) and of CHARG (which

FORTRAN CODING FORM

Program DEBT
Coded By Adam Smith
Checked By David Ricardo

Identification
DEBT
73 80

C FOR COMMENT

STATEMENT NUMBER 1 5	CONT. 6	FORTRAN STATEMENT 7
1		FORMAT (7E16.6)
		DEBT=300.00
		PAYMT=22.40
		CHARG=0.008*DEBT
		REDUC=PAYMT-CHARG
		RMAIN=DEBT-REDUC
		WRITE (3,1) DEBT, CHARG, PAYMT, REDUC, RMAIN
		STOP
		END

FIGURE 1: A FORTRAN program hand-written on a coding form. The form is shown in part only.

has been set in the fourth statement). The minus sign "-" is used for subtraction, the plus sign "+" for addition, the asterisk "*" for multiplication, and "/" for division.

The *sixth statement* computes the remaining debt RMAIN, as DEBT-REDUC.

The *seventh statement* tells the compiler to produce an object program, which *writes out* the contents of memory locations DEBT, CHARG, PAYMT, REDUC, and RMAIN, in that order. The numbers appearing inside the parentheses are *not* data numbers, and do *not* have decimal points. Their meaning is as follows: the number "3" is the number assigned to the line printer as its "unit number"; these numbers are discussed in Chapter IV. (*) The second number inside the parentheses, i.e., the number "1", is the statement number we have used for the FORMAT statement in the program.

Let us recall the values we worked out at the beginning: DEBT is 300.00, CHARG is 2.40, PAYMT is 22.40, REDUC is 20.00, and RMAIN is 280.00. We expect these five numbers to appear on one printed line, on the printed output page produced by the computer. So they do, but the appearance of these numbers is rather unfamiliar.

Each output number appears as *two* numbers, a fraction (called the "mantissa")

(*) Some installations assign other numbers to the line printer. Ask your installation for the unit number of the line printer; if it differs from 3, use that other number in place of the "3" in our WRITE statement.

and then another number (called the "exponent") which tells us by how many places the decimal point of the fraction has to be shifted, right or left as the case may be, to obtain the value which we really want. For example, the number 300.00 appears in the form:

0.300000E 03

This means we should take the mantissa, 0.300000, and then *move the decimal point three (03) places to the right.* The code symbol "E" stands for "exponent"; it shows where the mantissa ends and the exponent (number of shifts) starts. When we shift the decimal point three places to the right, we do indeed obtain the correct value, 300.000.

The entire line of output, containing the five numbers given before, reads:

0.300000E 03 0.240000E 01 0.224000E 02 0.200000E 02 0.280000E 03

You should make sure that you can read these numbers correctly as meaning 300.000, 2.40000, 22.4000, 20.0000, and 280.000, respectively.

If the number following the code symbol "E" is negative, then we must move the decimal point to the *left.* For example,

0.224000E-02

is to be read as 0.00224000. If the number following the code symbol "E" is 00, then the decimal point is not moved at all.

The advantage of this scheme is the ease with which very large and very small numbers can be written and read. For example, "0.118385E 32" takes less space to write, and is easier to read, than

11838500000000000000000000000000.0

Similarly, "0.118385E-34" takes less space to write, and is easier to read, than

0.00000000000000000000000000000000118385

The purpose of the first statement in our program (the FORMAT statement) is to tell the computer that the numbers are to be printed out in this particular form.

The *eighth FORTRAN statement* tells the compiler to produce an object program which *stops* program execution as soon as this point is reached. The supervisor takes over and proceeds to the next "job" in the "stream" of student jobs.

Important Note: In some installations, the normal command for stopping program execution is not STOP, but rather CALL EXIT. You should check with your own installation to find out which form they require. See also Chapter XVII.

The *last FORTRAN statement* of our program, END, is a signal to the compiler that nothing more remains to be translated from FORTRAN into machine language. *The END statement must be the last statement of every FORTRAN program.*

After the program has been written on the coding sheet, it must then be transferred to input cards, that is, to a form which the computer can accept. Input cards can be of two forms: punched cards and marked cards.

An example of a punched card, containing the first statement of our program, is shown on the opposite page. The card reader attached to the computer is sensitive only to the holes punched into the card. But for the convenience of human beings, the information appears also in typed form, near the top edge of the card.

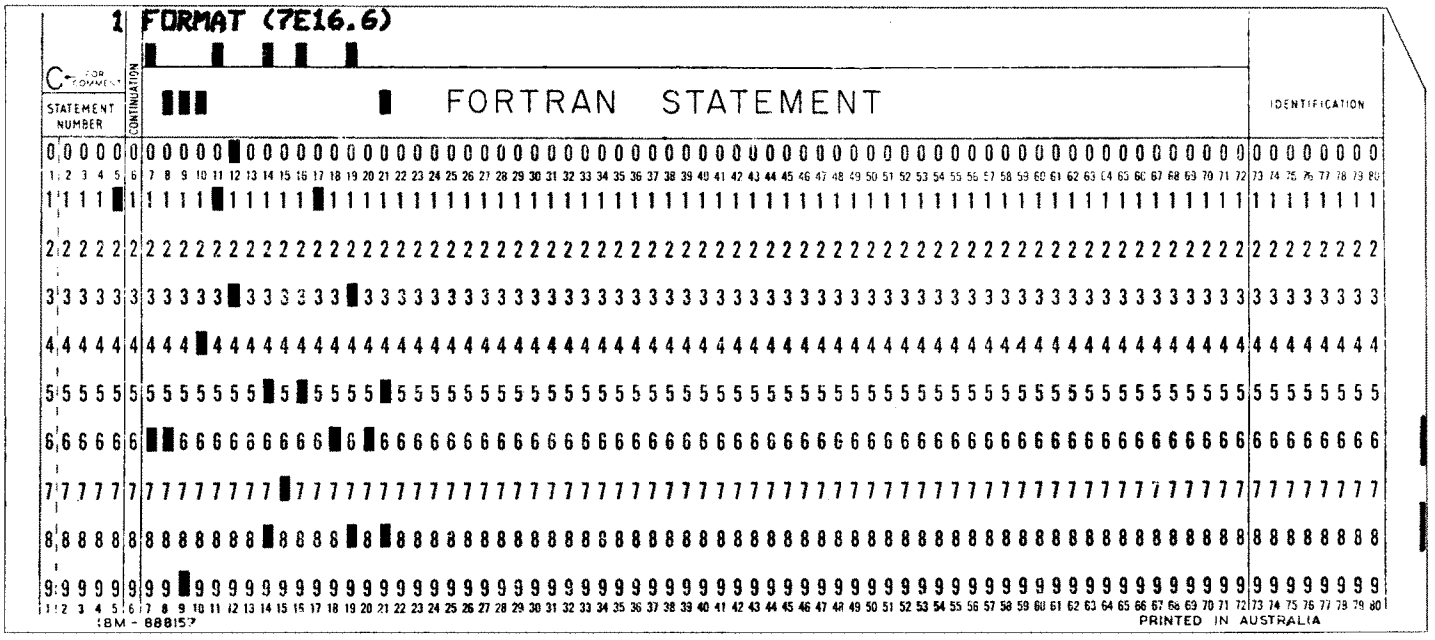


FIGURE 2: An example of a punched card, with a FORTRAN statement.

Note that each FORTRAN character is encoded into one column of the card, as a set of holes (it is *not* necessary for you to learn the code which is used). The punched card has eighty columns altogether. The last eight of those, that is, columns 73 to 80, inclusive, are not looked at by the FORTRAN compiler. They may be left blank, or they can be used for card identification (name of this program, sequence number of this card within this program).

The FORTRAN statement itself appears, starting in column seven (7) of the card. It may extend all the way to column seventy-two (72), inclusive. Column six (6) should remain blank (optionally, it may contain the character "0", but we advise against this).

In the rare cases that a FORTRAN statement is too long to fit entirely onto columns 7 through 72, inclusive, of one card, *continuation cards* may be employed. Columns one through five (1-5) of a continuation card *must* be blank. Column six (6) of a continuation card *must* contain some character (not a blank or a "0"). Starting in column seven, we have the continuation of the FORTRAN statement from the preceding card. No more than five (5) continuation cards are allowed for any one statement.

An initial card, but not a continuation card, may have a *statement number* punched into columns one through five (1-5). For example, the card shown in Figure 2 has a statement number equal to "1", punched into column five of the card.

The character "C" punched into column one (1) of the card indicates a *comment card.* The information on a comment card is reproduced (listed) on the output, but is not translated into machine language, and therefore does not need to follow the rules of FORTRAN grammar. A comment is simply a message to yourself, or to whoever else looks at the output, usually indicating what the program is meant to do here.

The second form of input card is the marked card, of which an example appears on the next page (with exactly the same FORTRAN statement).

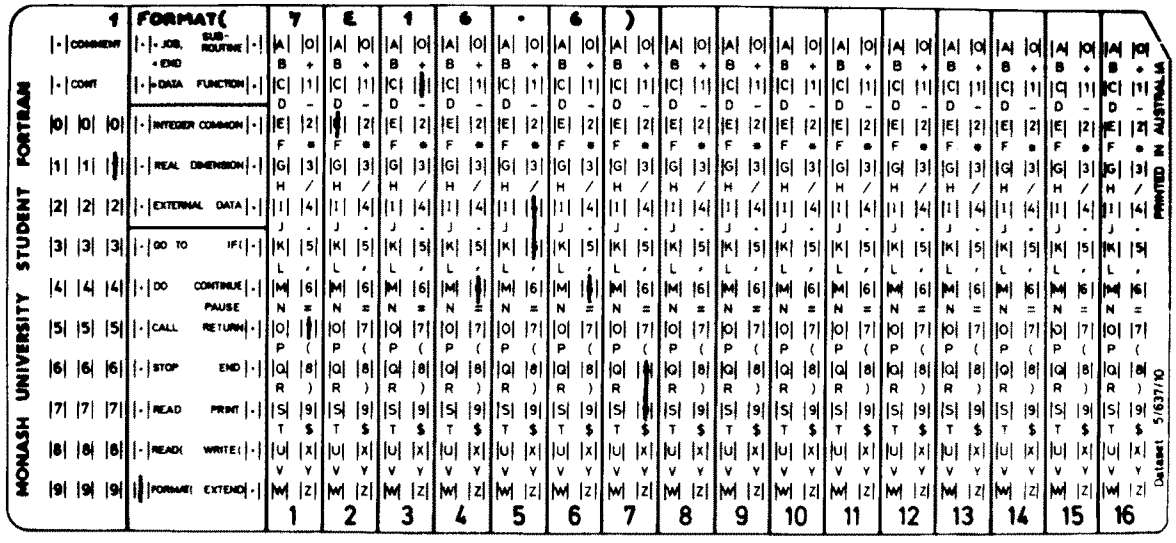


FIGURE 3: An Optically Marked Card.

No holes are punched into a marked card. An ordinary soft pencil is used to make black lines within the spaces provided on the card; these spaces may be indicated by parallel vertical lines, as above, or else may be enclosed by printed rectangles or printed ovals, within which the pencil marks are to be made. The special card reader attached to the computer reads black pencil marks in these spaces as if they were holes in a punched card. Often a special layout is used for the card, for example, in Figure 3 a single pencil mark suffices for the string of seven separate characters "FORMAT(". A marked card has less than eighty character positions; special techniques are employed to "piece together" two or more marked cards to give the equivalent of one eighty-column punched card.

A detailed discussion of card preparation techniques, suitable for your installation, appears in Chapter XIV, page 162. This Chapter can and should be read at this stage, before proceeding to prepare input cards for your programming exercises.

The program given on page 6 then appears as a little "card deck" of nine cards, *one card for each line written on the coding form.* Besides these nine FORTRAN cards, certain additional cards, called *control cards,* are required for every "job" run on the machine. Control cards are used to tell the computer whose job it is; that it is a program written in the FORTRAN language; etc. Control cards suitable for your installation are described in Chapter XV.

The entire deck of cards, including the FORTRAN language cards and the control cards, is called a "job deck". This job deck must be handed in, for running on the computer. The job deck is then returned to the programmer, and so is the printed output resulting from the run. A typical example of a complete printed output, from a particular FORTRAN compiler, is shown at the end of Chapter IV. A detailed discussion of the printed output produced by your particular version of FORTRAN appears in Chapter XVI. Briefly, the printed output starts with a reproduction ("listing") of all the input cards (control cards as well as FORTRAN language cards); this is followed by the output produced during the execution phase. This latter output is produced by the translated version of the WRITE statement in the program. In our example, this is only one printed line, containing five numbers in E-format; this line was shown on page 8, 11th line.

Summary and Formal Definitions for Chapter I

At the end of each Chapter, we give a brief and rather formal summary of the rules and definitions of the FORTRAN language which have been introduced in that Chapter. These summaries are not meant to be explanations; rather, they form a "manual" for easy and succinct reference. A complete "glossary" of the rules of USA Standard Basic FORTRAN IV, with page references to the appropriate explanations, appears on page 150.

1. *Character Set:*

   Numeric: 0 1 2 3 4 5 6 7 8 9

   Alphabetic: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   Special: Blank, equal "=", plus "+", minus "-", asterisk "*", slash "/", left parenthesis "(" right parenthesis ")", comma ",", period "."

   Alphameric: either alphabetic or numeric, but not special.

2. *Card Layout:*

   Comment card: "C" in column one, anything thereafter; not translated.

   END card: "END" in columns 7, 8, and 9 of the card; this must be the last card of a program.

   Statement cards: *Initial card* may have a statement number (between 1 and 9999) in columns 1-5, must have a blank or a "0" in column 6; the FORTRAN statement itself appears in columns 7 to 72, inclusive. *Continuation cards:* columns 1 to 5 must be blank, a character (other than a blank or a "0") must be in column 6; the continuation of the FORTRAN statement appears starting in column 7. Up to 5 continuation cards are permitted.

3. *Constants:*

   Integer constant: String of numeric characters, preceded by "+", "-", or blank; no decimal pt.

   Real constant: Must have a decimal point somewhere. The basic form is an ordinary real number with an explicit decimal point (e.g., "3.0" rather than just "3"). The E-form is a basic real number (called "mantissa"), followed by the letter "E", followed by an integer (called "exponent") which indicates the number of places that the decimal point is to be shifted (to the right if +, to the left if -).

4. *Identifier Names:*

   Between one and five alphameric characters, the first character must be alphabetic. A first character I, J, K, L, M, or N indicates integer values; all others are real number identifiers.

5. *Lists:*

   A simple list consists of identifier names, separated from each other by commas. There is no comma before the list, and no comma at the end of the list. Used so far only as a list of memory locations from which to take values to be printed out by a WRITE statement.

6. *Arithmetic Operators:*

   "+" for addition, "-" for subtraction, "*" for multiplication, "/" for division. If a multiplication is wanted, the multiplication sign "*" must be written explicitly.

7. *Types of FORTRAN Statements encountered so far:*

   Arithmetic assignment statement: (typical example) CHARG=0.008*DEBT
   The expression to the right of the "=" sign is evaluated, and the number so found is stored into the memory location named to the left of the "=" sign.

   WRITE statement: (typical example) WRITE (3,1) DEBT,CHARG,PAYMT,REDUC,RMAIN
   (general form) WRITE (*n,m*) *List*, where *n* is the number assigned to the printer in your installation, *m* is the statement number of a FORMAT statement, and *List* is a simple list of identifier names, naming the memory locations from which the values are to be "fetched" for printing on the output page.

FORMAT statment: (typical example) 1 FORMAT (7E16.6)
This kind of statement controls the way numbers are written out by a WRITE statement. For explanation, see Chapters IV and VII. Every FORMAT statement must have a statement number.

8. *Job Deck:*

The job deck is the set of cards necessary for running a "job" on the computer. It consists of: (1) Job card (sometimes followed by other control cards); (2) FORTRAN program cards, of which the last is the END card; (3) One or more control cards, signalling the start of the execution phase. These cards are described in detail in Chapter XV for your particular computer and FORTRAN compiler.

9. *A Note on Formal Definitions:*

In many formal definitions of FORTRAN statement types, we shall employ both ordinary letters and italics. For example, the general form of the WRITE statement was given as:

WRITE (*n,m*) *List*

with "*n*", "*m*", and "*List*" printed in italics. The things printed in ordinary type are always the same whenever this type of statement is used; for example the characters "WRITE" always appear, in every WRITE statement, in just that form. Things printed in italics, on the other hand, may change from one use of this statement type to the next use. For example, "*m*" is meant to be the number of a FORMAT statement. At the moment, we have only one FORMAT statement, to which we have given the statement number 1, so *m* is equal to 1 for the time being. But later on, we shall use WRITE statements referring to other FORMAT statements, with statement numbers different from 1.

There is a relationship between this kind of formal definition and the grammar of a language such as English. For example, the English sentences: "The man is old", "The cat is black", and "The world is flat" all have the common structure: "The *noun* is *adjective* ". In each sentence, a different noun takes the place of *noun*, and a different adjective takes the place of *adjective*. But the formal structure of the sentence is the same. The study of "formal grammars" is by now well-developed, and forms the basis for writing FORTRAN compilers.

DRILL EXERCISES FOR CHAPTER I

1. Which of the following are grammatically wrong real variable identifier names, and why?
KILLR, 123GO, K123, G123, DASTARD, ****, PROFESSOR, MAD, DOG, LOTTERY, LOTTRY, BE WARY, BEWARE, WHATONEARTH.

2. Which of the following numerical constants do *not* represent real numbers in FORTRAN, and why?
35. 35.000 35.001 .35000 35000. 35000.0 35,000 35,000. 35,000.0 35 -35 -35. -35.0

3. a) What does the character "C" in column 1 of a FORTRAN card mean?
b) What columns on a FORTRAN card are used for statement numbers?
c) What is the function of column 6 of a FORTRAN card?
d) In which column of the card does the FORTRAN statement itself start?
e) How many alphabetic characters are there? How many numeric characters?
f) Which of the following characters are alphabetic? Numeric? Special? Unacceptable?
2 (L 9 = T * % + ? 7 P Z -

4. Write the following numbers in ordinary decimal form:
0.786012E 04 0.786012E-04 -0.786012E 04 -0.786012E-04 0.786012E 00
-0.786012E 00 0.786012E-01 -0.786012E 01 -0.786012E-01 0.999999E 00
0.999999E 06

5. What is the main purpose of the FORTRAN compiler program? What is meant by "compile phase" and "execution phase"? What is the main advantage gained by using FORTRAN rather than machine language?

6. The printed output starts with a listing of the input cards, followed by output numbers. State which is produced during the compile phase, which during the execution phase.

7. For each of the following expressions, state whether it is valid FORTRAN; if not, why not? And if so, state the meaning of the expression.

a)	3A+%	b)	3A+5B	c)	A3+B5	d)	3*A+5*B
e)	3.0*A+5.0*B	f)	2.15*=A	g)	-2.15*A	h)	-A*2.15
i)	A/3	j)	A/3.0	k)	3/A	l)	3.0/A

8. A rectangular brick has side lengths A=5.1 inches, B=12.3 inches, and C=7.8 inches. Compute the volume of the brick by multiplying these three numbers together, store the result into memory location named VOLUM, and print out the values of A, B, C, and VOLUM.

9. Same as problem 8, but determine not only the volume but also the surface area, for which the formula is: 2(ab+bc+ca), and store this value into memory location AREA. Print out the values of A, B, C, VOLUM, and AREA, in that order.

10. A man receives an hourly wage of $2.25. He has worked 35 hours during the week. One fifth of his gross pay is withheld as a taxation deduction, and another tenth is withheld as a superannuation deduction. Compute and print out his gross pay GROSS, his taxation deduction TXDDC, his superannuation deduction SUDDC, and his take-home pay PAY.

11. Alter the program of problem 10 by including a savings deduction SVDDC, equal to four percent of the gross wage.

12. Same as problem 11, but allow for six hours of overtime OVTIM, paid at 1½ times the normal rate, in addition to the 35 hours of straight time.

PROGRAMMING EXERCISES FOR CHAPTER I

1. Same as drill exercise 8, but with a brick of dimensions 5.7, 12.8, and 8.6 inches, respectively.

2. Same as drill exercise 9, but with a brick of dimensions 5.7, 12.8, and 8.6 inches, respectively.

3. Same as drill exercise 10, but the man receives an hourly wage of $2.50, and has worked 40 hours that week.

4. Same as programming exercise 3, but include a savings deduction SVDDC, amounting to five percent of the gross wage.

5. Same as programming exercise 4, but allow for eight hours of overtime OVTIM, paid at 1½ times the normal rate, in addition to the 40 hours of straight time.

CHAPTER II

ARITHMETIC STATEMENTS

Section A: Initial Values, Assignment Statements.

At the start of the execution phase (see page 2 and 3) the contents of all the named memory locations in the machine are unknown (that is, they contain whatever has been left over from the preceding job in the job stream). Thus, before the content of a named memory location can be used, your program must set the initial value for the calculation. One method for doing so is the *assignment statement*, such as

```
AREA=5.89
```

In the execution phase, this instructs the machine to store the decimal number 5.89 into the memory location named by the identifier name AREA.

The equality sign "=" in this statement does *not* imply a mathematical equality. Rather, it represents an instruction to the machine to put the quantity to the right of the equality sign, into the memory location named to the left of the equality sign. For example, consider the statement:

```
AREA=AREA+2.5
```

In the execution phase, the machine is instructed to do the following:

1) Extract the present value of AREA from the memory location by that name;
2) Add the decimal number 2.5 to this (steps 1 and 2 together are called "evaluating the arithmetic expression to the right of the "=" sign");
3) Store this result into memory location AREA (thereby obliterating the number previously stored there).

If the two FORTRAN statements above appear as the first two statements of a FORTRAN program, then the content of memory location AREA is:

1) Unknown — at the time the execution phase starts;
2) 5.89 — after the first FORTRAN command has been obeyed;
3) 8.39 — after the second FORTRAN command has been obeyed;

The general arithmetic assignment statement has the form:

NAME=expression

where *NAME* is any grammatically correct identifier name, the symbol "=" is used to denote the assignment operation, and *expression* is any well-formed arithmetic expression (see the next Section). The effect of the assignment statement is, first, evaluation of the arithmetic expression, followed by storing the value so found into the memory location named by *NAME*.

Section B: Well-Formed Arithmetic Expressions.

We have already met several examples of arithmetic expressions, e.g.:

(1) A single number, such as the number 5.89 in the assignment statement AREA=5.89;

(2) The result of an arithmetic operation, such as the addition AREA+2.5 in the assignment statement AREA=AREA+2.5

More generally, *an arithmetic expression is an instruction to the machine to carry out some arithmetic operation or operations so as to get a result which is a number.* The arithmetic operations are denoted by:

Operation	*Sign*	*Example*	
Addition	+	A+B	means the sum of the values of A and B
Subtraction	-	A-B	means the difference between the values of A and B
Multiplication	*	A*B	means the product of the values of A and B
Division	/	A/B	means the value obtained by dividing A by B
Exponentiation	**	A**B	means the number obtained by raising the value of A to the power given by the value of B

If the current value of A is 6.0 and the current value of B is 2.0, then the value of A+B is 8.0, the value of A-B is 4.0, the value of A*B is 12.0, the value of A/B is 3.0, and the value of A**B is 36.0 (that is, 6 to the power 2, or 6 times 6, or 36). If B is altered to 4.0 (say, by the assignment statement B=2.0*B), then the new value of A**B becomes 1296.0 (that is, 6 to the power 4, or 6 times 6 times 6 times 6, or 1296).

If only a single arithmetic operation is desired, that is all there is to it. But if several arithmetic operations are called for, then it may be necessary to use parentheses to specify exactly what is wanted. When parentheses are used, then *operations enclosed within parentheses are carried out before the other operations.* For example, let us suppose that the current values of A, B, and C are A=24.0, B=2.0, and C=3.0, and let us consider the two arithmetic expressions (A/B)*C and A/(B*C):

(A/B)*C — *First* divide A by B, getting 12.0 as the value of the bracket.
Then multiply this value by C=3.0, getting 36.0 as the final result.

A/(B*C) — *First* multiply B by C, getting 6.0 as the value of the bracket.
Then divide this value into A=24.0, getting 4.0 as the final result.

Since the final results, 36.0 and 4.0, are completely different from each other, it matters a great deal in what order we carry out the arithmetic operations "/" and "*" here!

In order to avoid having to write large numbers of parentheses, there are a number of *parenthesis omission rules.* For example, when parentheses are omitted above, i.e., when we write simply "A/B*C", then the FORTRAN compiler assumes that the operations are to be carried out in the same order as reading the expression *from left to right:* the compiler translates this expression as if it had been (A/B)*C, not A/(B*C). If the latter is wanted, the parentheses must not be omitted.

If an expression without parentheses contains a number of different operation signs, then exponentiations "**" are carried out first; multiplications and divisions are carried out next (reading from left to right); and additions and subtractions are carried out last (again reading from left to right). We now give some examples, in which we assume the current values:

A=24.0, B=2.0, C=3.0, D=4.0:

1) A*B+C/D — *First* multiply A by B (left-most multiplication or division);
Then divide C by D (result = 0.75);
Then add these to get the final result = 48.75.
The fully bracketed form would be: (A*B)+(C/D)

2) A*(B+C)/D — *First* evaluate the bracketed part, getting B+C = 2.0+3.0 = 5.0;
The remainder is evaluated reading from left to right, hence:
Second, multiply A by 5.0, getting 120.0;
Last, divide this by D, getting the final result = 30.0.

3) A*(B+C/D) — We evaluate the bracketed part first of all; within the brackets, the division takes precedence over the addition; hence:
First, divide C by D (result = 0.75);
Second, add B to this, to get 2.75;
Last, multiply A by this 2.75, to get the final result = 24.0*2.75 = 66.0.

4) (A*B+C)/D — *First,* multiply A by B, to get 48.0;
Second (still within the bracketed part) add C to this, to get 51.0;
Last, divide this result by D, to get the final result = 51.0/4.0 = 12.75.

5) (A*(B+C))/D — Here we have brackets inside brackets. The innermost brackets are done first of all, i.e., we *start* by adding B and C to get 5.0;
Next, we evaluate the outer bracket, i.e., A*5.0, to get 120.0;
Last, we divide this by D to get the final result = 30.0.
Notice that the sequence of operations, as well as the final result, are exactly the same as for example (2). That is, the outer brackets were actually unnecessary, though they did no harm.

6) A*((B+C)/D) — *First,* evaluate the innermost bracketed expression, B+C, to get 5.0;
Next, evaluate the outer bracketed expression, 5.0/D = 1.25;
Last, do the multiplication outside of the brackets, to get 30.0.
Note that the sequence of operations differs from that in example (5), although the final result is the same. Note also that both the sequence of operations and the final result differ from those of example (3).

7) A+B**C*D — *First,* carry out the exponentiation: B**C = 2.0**3.0 = 8.0;
Next, carry out the multiplication: (B**C)*D = 8.0*4.0 = 32.0;
Last, carry out the addition: A+((B**C)*D) = A+32.0 = 56.0.

Parentheses must not be omitted if their omission would result in two arithmetic operation signs appearing right next to each other. For example, A*(-B) is grammatically correct FORTRAN, and instructs the machine to multiply the value of A by the negative of the value of B. But "A*-B" is grammatically incorrect (an "ill-formed" rather than a "well-formed" expression), because the two arithmetic operation signs "*" and "-" appear next to each other. Such an incorrect expression is not translated into machine language; rather, an error message ("diagnostic") is issued by the compiler onto the printed output.

For the purpose of the rule which we have just given, the exponentiation (raising to a power) sign "**" is treated as a single operator. That is, "**" is not considered to be two multiplication signs next to each other, and "**" is valid. But "***" would be an ungrammatical form, and would give rise to a diagnostic message.

It is permissible to start an expression with a minus sign; e.g., "-A*B" is correct FORTRAN and means the negative of the product of A and B.

Section C: Real Numbers and Integers. Mixed Mode Expressions. Integers as Powers.

We now give a brief explanation of two rules enforced so far:

1. All numbers must be written with a decimal point, e.g. "3.0" rather than just "3";
2. Identifier names must not start with one of the letters: I, J, K, L, M, or N.

There are two kinds of numbers in daily use that are different, even though most of us pay little attention to the difference. These two kinds of numbers, however, are stored in different ways inside a computer, and are treated differently by the computer. The two kinds of numbers are *integers* and *real numbers.*

Integers, or counting numbers, have no fractional part. It makes no sense to speak of page number 12.73 of a book, or of a class with 43.28 students. When two integers, for example, 12 and 13, are added, subtracted, or multiplied, the result is again an integer. But division is not always possible; for example, 12/13 is not an integer, and neither is 13/12.

Real numbers, or measuring numbers, enable us to express fractional parts. A stick may be exactly 27 inches long. But it could just as easily have a length of 27.5 or 26.9 inches. If a stick of length 27 inches is broken into two equal halves, then each half has a length of 13.5 inches, not 13 inches or 14 inches. Thus the fractional parts are important, and division always gives a sensible result (except for division by zero!).

In FORTRAN, variable identifier names starting with any of the letters I, J, K, L, M, or N are reserved for quantities whose values are integers. Similarly, constant numbers written without the use of an explicit decimal point are treated as integer numbers in the machine.

The use of integers, and the rules for arithmetic operations with integers, will be discussed in Chapter VI. But two matters must be discussed now, namely *mixed mode expressions* and *integers as powers.*

An arithmetic expression which requires the machine to add, subtract, multiply, or divide two numbers, one of which is an integer, the other a real number, is said to be an expression of mixed mode. Such expressions are *forbidden* in USA Standard Basic FORTRAN IV.

Examples:

3*A	is forbidden	3.0*A	is allowed	3*K	is allowed
A+4	is forbidden	A+4.0	is allowed	K+4	is allowed
3.7-2	is forbidden	3.7-2.	is allowed	4-2	is allowed
A/I	is forbidden	A/G	is allowed		

Note: Some "dialects" of FORTRAN permit the use of mixed mode expressions. In those dialects, all integers appearing in mixed mode expressions are converted to real number form inside the machine, before the expression is evaluated. Since these conversion operations take machine time, and since mixed mode expressions will work only with some, not with all, existing FORTRAN compilers, mixed mode expressions should be avoided.

Integer constants (strings of numeric characters without a decimal point) should not be used at this stage. However, there is one exception to this rule: *Integer constants should be used as exponents whenever an integral power is wanted.* For example:

X**3 is greatly preferable to X**3.0

Explanation: Although these two expressions are equivalent mathematically, they are translated by the compiler into two quite different sets of machine language commands. X**3 means X*X*X; that is, the machine is instructed to multiply the value of X by itself, getting (X*X); and to multiply this result once more by X, thus getting (X*X)*X. On the other hand, when we specify "X**3.0", the machine is instructed to take the logarithm of X (don't worry if you have forgotten about logarithms; it does not matter for this discussion), then to multiply this logarithm by the real number 3.0, and then to take the anti-logarithm of the result. This second procedure, which we shall not bother to explain, has a number of disadvantages: (i) it takes much more machine time; (ii) it gives lower accuracy, and (iii) it cannot be done at all if the value of X happens to be negative!

In simple problems, whenever powers of numbers are wanted at all, they are likely to be integral powers. In FORTRAN programs, the exponents should then be written as integer numbers, or integer identifier names.

Summary and Formal Definitions for Chapter II.

1. *Assignment Statements:*

   General form: *NAME=expression*

   where *NAME* is an identifier name, and *expression* is a well-formed arithmetic expression.

2. *Well-formed Arithmetic Expression* (fully bracketed):

   We abbreviate "well-formed arithmetic expression" by "wfae"; the rules by which such an expression is defined are:

   (1) A constant is a wfae.

   (2) An identifier name is a wfae.

   (3) If *W* is a wfae, then (-*W*) is also a wfae, whose value is the negative of the value of *W*. The parentheses may be omitted in a number of cases (see parentheses omission rules, later).

   (4) If *W* and *V* are wfae of the same type (both integer or both real), then (*W*+*V*) is itself a wfae, whose value is the sum of the values of *W* and of *V*.

   (5) If *W* and *V* are wfae of the same type, then (*W*-*V*) is itself a wfae, whose value is the difference between the value of *W* and the value of *V*.

   (6) If *W* and *V* are wfae of the same type, then (*W***V*) is itself a wfae, whose value is the product of the values of *W* and of *V*.

   (7) If *W* is a wfae of real number type, and *V* is a wfae of either integer or real number type, then (*W****V*) is itself a wfae, whose value is the value of *W* raised to the power given by the value of *V*. If *V* is of real number type, then the value of *W* must not be negative. If *W* and *V* are both of integer type, the (*W****V*) is itself a wfae, whose value is the value of *W* raised to the power given by the value of *V*.

   (8) If *W* and *V* are wfae, both of type *real,* then (*W*/*V*) is itself a wfae, whose value is the quotient obtained by dividing the value of *W* by the value of *V*.
   If *W* and *V* are wfae, both of type *integer,* then (*W*/*V*) is itself a wfae, whose value is obtained by dividing the value of *W* by the value of *V*, and then omitting all digits after the decimal point of the quotient, thereby obtaining another integer. (See page 59).

3. *Parentheses Omission Rules:*

   For the purpose of stating the parentheses omission rules, the arithmetic operations are arranged in a rank order, ranging from the highest rank "3" to the lowest rank "1". This rank order is:

Rank	*Operations*	*Rule no.*
3	**	(7)
2	*	(6)
	/	(8)
1	+	(4)
	-	(3,5)

Parentheses omission rule 1: Unless specified otherwise by use of parentheses, operations of superior rank are performed before operations of inferior rank.

Parentheses omission rule 2: Unless specified otherwise by use of parentheses, operations of rank 2 are performed in the same sequence as reading the expression *from left to right;* operations of rank 1 are performed in the same sequence as reading the expression *from left to right;* parentheses must not be omitted from expressions such as (A**B)**C and A**(B**C), which two expressions give quite different results.

Parentheses omission rule 3: The outermost parentheses of an arithmetic expression can be omitted altogether.

Parentheses omission rule 4: If omission of a pair of parentheses would result in two arithmetic operation signs appearing right next to each other, then the parentheses must be retained.

Note: When in doubt about the sequence of operations in the absence of parentheses, it is best to retain the parentheses. However, large numbers of parentheses tend to cause errors, such as omission of a final right parenthesis, or insertion of an extra parenthesis. It is worth the effort to become familiar with the parentheses omission rules!

Note: The following technique is of considerable help in checking that the parentheses match properly. Read the expression from left to right, and compute a "bracket count" as follows: At the equal sign, the bracket count is zero; thereafter, the count increases by one whenever a left-hand parenthesis "(" is encountered; the count decreases by one whenever a right-hand parenthesis ")" is encountered. The bracket count must never become negative, and it must return to zero after the whole expression has been read.

DRILL EXERCISES FOR CHAPTER II.

1. Explain why the FORTRAN symbol "=" is not a mathematical equality sign, and state its meaning in FORTRAN.

2. A FORTRAN program starts with the three statements:

```
GUFF=14.4
GUFF=GUFF/3.0
GUFF=GUFF-GUFF
```

State what is known about the content of memory location GUFF:

(a) At the start of the execution phase;
(b) After execution of the first statement;
(c) After execution of the second statement;
(d) After execution of the third statement.

3. Write FORTRAN commands to do the following:

(a) Find the sum of A and B, and store it into location C.
(b) Find the sum of A and B, and store it into location A.
(c) Subtract 1.5 from A, multiply the result by B, store final result into C.
(d) Subtract 1.5 from B, multiply the result by A, store final result into A.
(e) Multiply A by B, subtract 1.5 from this, store the final result into C.
(f) Find the square of A, find the cube of B, add them, store the sum into C.
(g) Find the square of A, find the cube of B, multiply them, store the result into C.
(h) Find the sum of A and B, multiply by the difference between A and B; store result into C.

4. Assume that memory locations A, B, C, and D contain the numbers A=12.0, B=3.0, C=4.0, and D=6.0, respectively. For each FORTRAN statement below, state the sequence of arithmetic operations to be performed, and the numerical value placed into memory location F:

(a)	F=A*B+C	(b)	F=A*(B+C)	(c)	F=(A*B)+C
(d)	F=A*B/C	(e)	F=(A*B)/C	(f)	F=A*(B/C)
(g)	F=A*(B+C/D)	(h)	F=A*B**2	(i)	F=(A*B)**2
(j)	F=A/B**2	(k)	F=(A/B)**2		

5. Inspect the following arithmetic statements for errors:

(a)	F=3*A+B	(b)	F+2=A+B	(c)	F+2.0=A+B
(d)	F=A*35+B	(e)	F=35A+B	(f)	F=35.0A+B
(g)	F=MAN+BOY	(h)	F=(A+3*B))	(i)	F=((A+3)*B)
(j)	F=((A+3.0*B)	(k)	F=(A**2+B**3)/(C+4)	(l)	F=(A**2+B**3)/C**4
(m)	F=A+B*-C	(n)	F=-C*B+A	(o)	F=A+17.5B
(p)	F=HIGH+14.7WIDTH	(q)	F=HIGH+14.7*SIDELNGTH		

6. Write FORTRAN arithmetic assignment statements for the following (use 3.14159 for p, and 9.801 for g):

(a)	$s=\frac{1}{2}gt^2$	(b)	$V=4pR^3/3$	(c)	$e=iR$
(d)	$h=v^2/2g$	(e)	$v=\sqrt{2gh}$	(f)	$A=pR^2$
(g)	$R=\sqrt{(A/p)}$				

CHAPTER III

PROGRAM FLOW CONTROL

Section A: Statement Numbers; the GO TO Statement.

Normally, the computer obeys the (translated) commands in a FORTRAN program in the order in which they appear in the input card deck, card after card, until the last command is reached. There are many occasions when we want to interrupt this "normal flow" of program execution, to order the computer to obey some other command next.

To do this, we have to give "labels" to some of the FORTRAN statements. In FORTRAN, these take the form of numbers, which are called either *statement numbers* or *statement labels* (the two terms are used interchangeably).

To label a FORTRAN statement, the statement number (an integer between 1 and 9999, inclusive) must appear on the input card, anywhere in columns one through five of the initial card of the statement. Continuation cards (if any) carry no statement number, rather, columns one through five of a continuation card must be blank. No two statements are allowed to have the same statement number. Leading zeros make no difference; for example, statement number "0001" is the same as statement number "1".

Notes:

1. The statement number by itself has no influence on the sequence in which the machine obeys the FORTRAN commands. In the absence of specific instructions to the contrary, the commands are still obeyed in the order in which they appear in the input card deck, not in the order of increasing statement numbers.
2. It is conventional (but not strictly necessary) to write statement numbers in such a way that the last digit of the number appears in column five of the card. This is called "right-adjusted" within the "field" consisting of columns one through five of the card. ("Left-adjusted" would mean that the first digit of the number appears in column one; this is permitted, but is not the usual choice).

For example, suppose that we wish to number the assignment statement on the fourth line of the program on page 6, that is, the statement:

```
CHARG=0.008*DEBT
```

with the statement number "1500". The card must be altered to read:

```
1500 CHARG=0.008*DEBT
```

where the last digit of the number "1500" appears in column five of the card, column six is left blank, and the first letter "C" of "CHARG" appears in column seven.

The program on page 6 computes relevant numbers for only the first repayment on the debt. Suppose we want the machine to compute, and print out, a complete table

of all repayments. We could write out a new set of orders, for the second repayment, by starting from the debt RMAIN remaining after the first repayment. We could then write out still a new set of orders to compute quantities for the third repayment, and so on. Obviously, this would be most awkward and wasteful, and it would be easier to do the whole computation by hand.

In fact, however, all these computations differ from each other in only one respect: the current value of the unpaid debt DEBT. So all we wish to do is to go repeatedly through the same set of steps, replacing the current value of DEBT by its new value after each month's repayment. The new value of DEBT, in each case, is the preceding value of RMAIN. For example, after the first repayment, the value of RMAIN turned out to be 280.00 (see the bottom of page 5). The new value of DEBT at the start of the second month is equal to this 280.00.

To achieve our aim, we shall replace the statement "STOP" in the program by something which does not stop the machine, but rather which sets up the calculation for the next repayment. To do this, the revised program reads:

```
    1 FORMAT (7E16.6)
      DEBT=300.00
      PAYMT=22.40
 1500 CHARG=0.008*DEBT
      REDUC=PAYMT-CHARG
      RMAIN=DEBT-REDUC
      WRITE (3,1) DEBT,CHARG,PAYMT,REDUC,RMAIN
      DEBT=RMAIN
      GO TO 1500
      END
```

Explanation:

Let us follow through what this program does. The first time through, we do exactly as in Chapter I; when we reach the WRITE statement, we print out one line, containing the numbers 300.00 for DEBT, 2.40 for CHARG, 22.40 for PAYMT, 20.00 for REDUC, and 280.00 for RMAIN. (See line number 11 on page 8).

The statement DEBT=RMAIN, immediately following the WRITE statement, has the effect of "fetching" the contents of memory location RMAIN (that is, the number 280.00) and storing this value into memory location DEBT, thereby overwriting the previous value of DEBT. The new value of DEBT is therefore 280.00, which is the correct value at the beginning of the *second* month. Thus, we have "updated" the value of DEBT successfully, and we are ready to calculate the second repayment.

The *unconditional transfer of control* statement "GO TO 1500" tells the machine that the command with statement number 1500 must be obeyed next. This sends us to the point where we compute a new value of CHARG, this time based on a DEBT of 280.00; this new value of CHARG turns out to be 2.24; next, we compute a new value of REDUC, equal to 22.40-2.24=20.16, and a new value of RMAIN, equal to 280.00-20.16=259.84. We then write out the next line of the table, which contains the numbers 280.00, 2.24, 22.40, 20.16, and 259.84, in that order.

Next, we fetch the current contents of RMAIN, 259.84, and store this value into memory location DEBT. This is the correct DEBT at the beginning of the *third* month. We return to statement number 1500 once more, to commence the calculations for the third repayment on the debt.

We print out line after line of output, one line for each repayment. The first few lines of output from this program are shown below (for reasons which will be explained soon, we do not reproduce the entire output from this program).

0.300000E 03	0.240000E 01	0.224000E 02	0.200000E 02	0.280000E 03
0.280000E 03	0.224000E 01	0.224000E 02	0.201600E 02	0.259840E 03
0.259840E 03	0.207872E 01	0.224000E 02	0.203213E 02	0.239519E 03
0.239519E 03	0.191615E 01	0.224000E 02	0.204838E 02	0.219035E 03
....	....	....	....	....

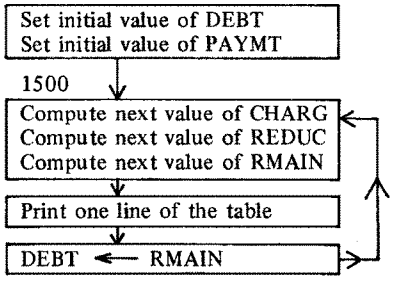
It is neither necessary nor desirable to assign a statement number to every FORTRAN statement in a program. Statement numbers *must* be assigned to:

1. Statements to which control transfers take place; and
2. Statements immediately following a GO TO statement.

When a statement follows immediately after a GO TO statement, the "normal flow" of control is diverted away from it by the GO TO statement. If the statement is not labelled by a statement number, then there is no way to jump (transfer control) to it. Thus our unfortunate statement of type 2 can not be reached at all during program execution.

Section B: Loops, Flow Diagrams, and Conditional Transfers of Control.

The program just given may look all right, and the first few lines of output are quite correct. But there is one thing very wrong with it. To see what it is, let us construct a *flow diagram* for this program, similar to the flow diagram on page 3.



There is clearly a "loop" of commands, extending from statement number 1500 down to, and including, the statement GO TO 1500. The machine goes through this loop over and over again, each time doing a new computation and printing out a new line of the table.

But there is the trouble! *Nothing in the program tells the machine to stop!* It goes through the same loop again, and again, and again, forever and ever. After a while, the debt has been more than paid off, the current value of DEBT in the machine is already negative (meaning that the finance company now owes money to the customer, not the other way round). But still the machine keeps going, blindly, stupidly, and utterly devoid of sense, doing exactly what it has been instructed to do. It has never been told to stop, hence it does not stop! (*)

Let us see whether we can arrange to stop the machine after the debt has been

(*) Actually, the "supervisor" (see page 3) does not permit indefinite looping of this sort. Student programs are cut off after they exceed a preset time limit. But this is not the point here.

paid off. We can tell when we have reached this point, by looking at the current value of RMAIN. As long as RMAIN is a positive number, we have a remaining unpaid debt, and program execution should continue. As soon as RMAIN becomes zero or negative, we have arrived at the last repayment. We would therefore like a FORTRAN command which inspects the content of a memory location, and allows us to do different things (take different paths of control) depending upon whether the number in that memory location is negative, zero, or positive.

This FORTRAN command exists, and is called the IF statement. There are three statement numbers listed in an IF statement, as well as an arithmetic expression of some sort, call it *W*. We transfer control to the first numbered statement if the value of *W* is negative; to the second numbered statement if the value of *W* is zero; and to the third numbered statement if the value of *W* is positive.

Examples:

1. The statement "IF(RMAIN) 2200,2500,2000" has the effect of testing the numerical value stored in memory location RMAIN. Control is transferred to statement number 2200 if RMAIN is negative at this moment; to statement number 2500 if RMAIN is zero at this moment; and to statement number 2000 if RMAIN is positive at this moment.

2. The statement "IF(A-B) 1450,1270,1320" has the effect of testing the difference A-B. We are sent to statement number 1450 if A-B is negative at this moment (i.e., if A is *less than* B); to statement number 1270 if A-B is zero at this moment (i.e., if A is *equal to* B); and to statement 1320 if A-B is positive at this moment (i.e., if A is *greater than* B).

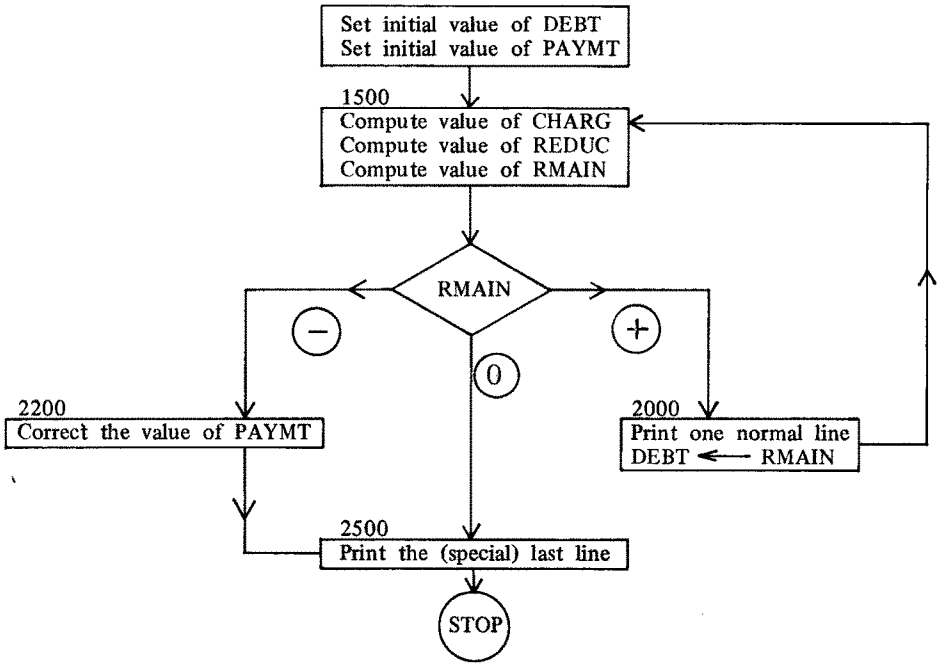
Let us now return to our infinitely looping program, or rather to its flow diagram given on page 23 and ask: at what point should we insert the conditional transfer statement testing the value of RMAIN? It must be *after* a new value of RMAIN has been computed, but *before* this value is printed out (since the value could be negative, and thus subject to correction before printing). Looking at the flow diagram, this narrows it down completely: the IF statement has to come below the box labelled "1500" and above the box saying "Print one line of the table".

We shall represent a *conditional transfer command* by a diamond-shaped "box" containing the quantity to be tested, and having three lines leading away from it: the line marked "-" is the path of control taken if the quantity is negative; the line marked "0" is the path of control taken if the quantity is zero; and the line marked "+" is the path of control taken if the quantity is positive.

The corrected flow diagram is given opposite, and the FORTRAN program immediately after it. Note that we must supply statement numbers for all statements to which transfers of control take place. In practice, these statement numbers would be left off the flow diagram initially, and supplied while writing the code based on this flow diagram.

Although it is not necessary to keep statement numbers in increasing sequence throughout the program, this is good clean coding practice, since it tends to prevent the assignment of the same statement number to two different statements.

It is also clean coding practice to use statement numbers which are spaced widely apart from each other, in case additional statement numbers must be found for statements in between the numbered ones. Clean coding is not just an affectation. Rather, it is an enormous help later on, in finding the inevitable bugs in the program.




```
    1 FORMAT (7E16.6)
      DEBT=300.00
      PAYMT=22.40
 1500 CHARG=0.008*DEBT
      REDUC=PAYMT-CHARG
      RMAIN=DEBT-REDUC
      IF(RMAIN) 2200,2500,2000
C     WE GET HERE IF THE REMAINING DEBT IS STILL POSITIVE
 2000 WRITE (3,1) DEBT,CHARG,PAYMT,REDUC,RMAIN
      DEBT=RMAIN
      GO TO 1500
C     WE GET HERE IF RMAIN WOULD HAVE TURNED OUT NEGATIVE
 2200 PAYMT=DEBT+CHARG
C     THIS IS THE CORRECT VALUE FOR THE LAST REPAYMENT
C     NOW PROCEED TO PRINT OUT THE LAST LINE
 2500 WRITE (3,1) DEBT,CHARG,PAYMT
      STOP
      END
```

The output numbers from this program are given on the next page. We see that the output does indeed come to an end after the debt has been paid off. All but the last line of the table are produced by the WRITE statement in the interior of the loop, statement number 2000. This has a list of five quantities to be printed out, and thus all but the last line of the table contain five numbers. The last line of output is produced by WRITE statement number 2500, which has a list of three quantities; thus the last line of the table contains only three numbers, namely the contents of the three memory locations DEBT, CHARG, and PAYMT, respectively. Naturally, the last value of PAYMT is just enough to retire the debt completely, i.e., the last PAYMT is equal to the sum of DEBT and CHARG at this point. If RMAIN had happened to be exactly zero at this stage, the last payment would have been the same as all the others, namely 22.40. Actually RMAIN turned out to be negative, so we went to statement 2200 to recompute the last payment, which then turned out to be 5.00 (less than the others).

0.300000E 03	0.240000E 01	0.224000E 02	0.200000E 02	0.280000E 03
0.280000E 03	0.224000E 01	0.224000E 02	0.201600E 02	0.259840E 03
0.259840E 03	0.207872E 01	0.224000E 02	0.203213E 02	0.239519E 03
0.239519E 03	0.191615E 01	0.224000E 02	0.204838E 02	0.219035E 03
0.219035E 03	0.175228E 01	0.224000E 02	0.206477E 02	0.198387E 03
0.198387E 03	0.158710E 01	0.224000E 02	0.208129E 02	0.177574E 03
0.177574E 03	0.142059E 01	0.224000E 02	0.209794E 02	0.156595E 03
0.156595E 03	0.125276E 01	0.224000E 02	0.211472E 02	0.135447E 03
0.135447E 03	0.108358E 01	0.224000E 02	0.213164E 02	0.114131E 03
0.114131E 03	0.913049E 00	0.224000E 02	0.214869E 02	0.926441E 02
0.926441E 02	0.741153E 00	0.224000E 02	0.216588E 02	0.709853E 02
0.709853E 02	0.567883E 00	0.224000E 02	0.218321E 02	0.491532E 02
0.491532E 02	0.393226E 00	0.224000E 02	0.220068E 02	0.271465E 02
0.271465E 02	0.217172E 00	0.224000E 02	0.221828E 02	0.496364E 01
0.496364E 01	0.397091E-01	0.500335E 01		

Note: Although the conditional transfer statement tests explicitly whether an arithmetic expression has the value zero, this can easily give misleading results, and must be used with caution. The trouble is that real numbers in the machine are not exact, but are carried only to a limited number of decimal places (the number of decimal places carried depends on the computer on which the work is done; the accuracy carried during the computation is not part of the specifications of the FORTRAN language as such: see Chapters VI and XVII). When arithmetic operations are carried out with such numbers of limited accuracy, the result is in general only approximate, not exact. For example, suppose A is 1.2 and B is 2.3; then the sum A+B in the machine may turn out to be 3.499997 rather than exactly 3.500000. The conditional transfer statement

```
IF(A+B-3.5) 2300,2400,2500
```

would then send us to statement number 2300 rather than to statement number 2400, merely because of these "round-off errors". Only in integer arithmetic (see Chapter VI) can one be sure of exact results. (*)

Section C: Two-Way Branching.

The definition of the IF statement is in the form of a three-way branching of control. Often we require only a two-way branch. For example, we may wish to go to statement 2000 if A is *greater than* B, and to statement 2300 otherwise. This can be achieved by writing "IF(A-B) 2300,2300,2000". Similarly, suppose we want to transfer control to statement 2000 if and only if A is *greater than or equal to* B. Then we write the command "IF(A-B) 2300,2000,2000".

In general, whenever two of the three statement numbers in the IF statement are equal, the IF statement gives rise to a two-way branch. If all three statement numbers are equal, the IF statement becomes equivalent to an unconditional transfer; e.g., the command "IF(A-B) 2000,2000,2000" transfers control to statement 2000 no matter what the values of A and B are, and is therefore equivalent to "GO TO 2000".

One form of two-way branching which occurs frequently in practice is a test whether

(*) This round-off error problem does not alter the logic of our program here, however. We want to do something different when RMAIN is exactly zero, even if this exact zero value involves round-off errors.

some number, say X, lies within a specified range. For example, we may wish to go to statement number 2000 if X lies between 1.6 and 4.8, and to go to statement number 1500 if X is equal to or less than 1.6, or if X is equal to or greater than 4.8. One way to achieve this is by a sequence of two two-way branches, first testing X-1.6, then testing 4.8-X; we transfer to statement 2000 if and only if both these are positive:

```
      IF(X-1.6) 1500,1500,1200
1200  IF(4.8-X) 1500,1500,2000
```

You should check through this code yourself, to see that it does what we want.

However, there is an easier way to achieve the same thing. Let us consider the product: (X-1.6)*(4.8-X). If X is less than 1.6, this product is negative; if X lies between 1.6 and 4.8, the product is positive; if X exceeds 4.8, the product is once more negative; finally, if X is either 1.6 or 4.8, exactly, then the product is zero. We can therefore achieve the same result by a single IF statement:

```
      IF((X-1.6)*(4.8-X)) 1500,1500,2000
```

Not only is this simpler FORTRAN coding, but on many machines it will be translated into a sequence of commands which is obeyed in less machine time than the more complicated coding.

Note the double parentheses in the above IF statement. According to the definition of the IF statement, the arithmetic expression to be tested must be enclosed within parentheses. In our case, the expression to be tested is "(X-1.6)*(4.8-X)", where the brackets are necessary for the arithmetic expression itself, to get the right result (can you tell what the result would be in the absence of these parentheses?). This entire expression must now be enclosed within an outer set of parentheses, to satisfy the FORTRAN language rules for an IF statement.

Note also that FORTRAN has only one kind of parentheses, whereas in mathematics we often employ several different symbols (round parentheses, square brackets, curly brackets, etc.).

Another, related, test is for the numerical value of some number, irrespective of sign (the so-called "absolute value" of a number). Suppose we want to go to statement 1500 if the absolute value of X is less than or equal to 1.6, and to go to statement 2000 if the absolute value of X exceeds 1.6. It is then useful to test the value of X**2, the square of X, against 2.56, the square of 1.6. X**2 is positive no matter what the sign of X. We therefore test the absolute value of X against 1.6 by means of:

```
      IF(X**2-2.56) 1500,1500,2000
```

However, a quicker method for testing an absolute value exists, see page 96.

Section D: More About Loops; Debugging.

Certain features are common to all program loops. There is always some quantity, such as the variable DEBT in our example, whose value decides whether we continue going around the loop, or exit from the loop. This quantity is called the *loop variable.* This loop variable must be *preset* before the loop itself is entered. In our example, we had the statement "DEBT=300.00" before entering the loop. Within the loop, the loop variable must be *updated* (by "DEBT=RMAIN"), and it must be *tested* (by means of the IF statement, which tests RMAIN, the next value of DEBT).

The loop in our previous example is slightly unusual, in the sense that the termination test involves a true three-way branch. Usually, the loop termination test is only a two-way branch, the decision being whether to continue the loop, or jump out of it.

We now present another example of a loop, this time with a two-way branch in the termination test. The problem we wish to solve is:

Find the square root of a given number, accurate to six significant figures.

The method we shall use is the following: Let "p" be a guess at the square root of the given number "a". Then a better estimate of the square root is given by the formula:

$$q = \frac{1}{2}(p + \frac{a}{p})$$

To start with, let us test this scheme with the particular given number a=9.6. We can look up the square root of 9.6 in a table of square roots, to find the correct answer:

$\sqrt{9.6} = 3.0983867$ (correct answer to eight significant figures)

Let us see how well our scheme works when we start with a guess p=3.0, say. The formula gives:

$$q = \frac{1}{2}(3.0 + \frac{9.6}{3.0}) = 3.100000$$

Note the great improvement: The initial guess p=3.000000 differs from the true answer by -0.0984 (the minus sign means 3.000000 is lower than the true answer); the improved estimate q=3.100000, on the other hand, differs from the true answer by only +0.001613. That is, the error (0.0016) in the estimate q is 61 times smaller than the error (-0.0984) in the initial guess p! Quite an improvement, that, for very little actual work!

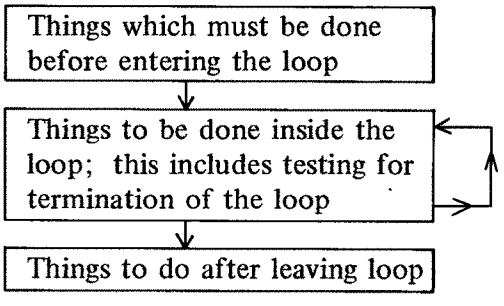
We can now get even closer to the true answer, by using the improved value 3.1 as the "p" in the formula (this is called an "iteration"). This gives the second estimate:

$$q = \frac{1}{2}(3.1 + \frac{9.6}{3.1}) = 3.0983871$$

The error in this number (difference from the true result) is only 0.0000004. That is, we now have more than the six correct significant figures demanded by the statement of the problem. The "3" is the first significant figure, and the "1" at the end of q is the eighth significant figure. Only this last figure is incorrect; the "7" before it is *right,* since the correct answer, *rounded to this number of places,* would be written as 3.098387.

Let us now draw a flow diagram for a FORTRAN program to do the same calculation (in Chapter IV, we shall show how this program can be altered slightly to enable us to calculate the square root of any number, not just the number 9.6).

A flow diagram for a program with a loop in it always has at least three "boxes", as follows:



An inexperienced programmer tends to start drawing the flow diagram from the top down, taking the first box first. This is poor practice, and tends to waste a lot of effort. It is much better to start with the middle box, i.e., to start by deciding just what it is we must do within the loop. The top and bottom boxes should be drawn in only later on.

Within the loop, we must use the formula to compute an improved estimate Q from the previous estimate P. The FORTRAN statement for doing so is:

```
Q=0.5*(P+A/P)
```

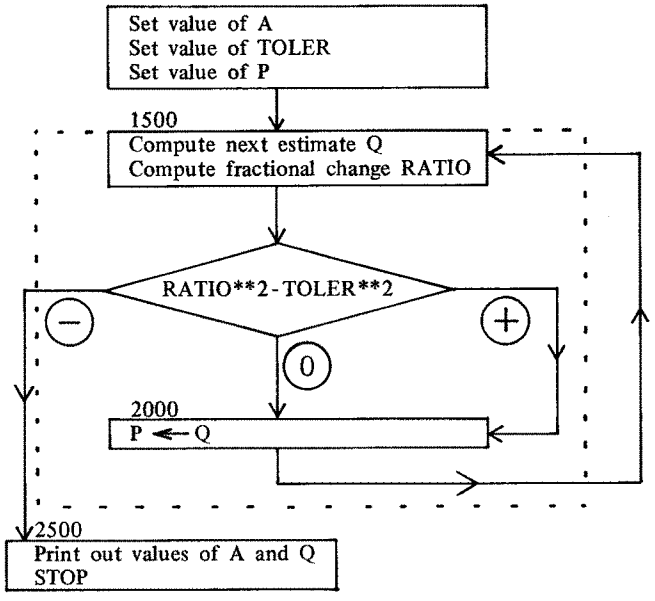
Next, we must test whether this new estimate of the square root is good enough. One way to do so is to look at the "fractional change from P to Q", i.e., at the ratio:

```
RATIO=(P-Q)/Q
```

If the absolute value of this ratio is less than the tolerance TOLER=0.000001, meaning one part in a million, then we have the accuracy demanded by the problem. Otherwise, we must go around the loop some more. To repeat the loop, we must put the value of Q into the memory location named P, by the statement "P=Q", and we must transfer control back to the beginning of the loop.

Having decided what we do inside the loop, let us now fill in the top and bottom boxes of the flow diagram. In the top box, we must set the value of A, the value of the tolerance TOLER, and a value for the first guess P. In the bottom box, we must write out the answer and, preferably, also the value of A for which the answer is the square root; and we must not forget to STOP program execution!

The flow diagram based on these considerations appears below. But we shall say a bit more before writing down the FORTRAN code.



We have indicated the middle box of the schematic flow diagram before, i.e., the interior of the loop, by drawing a dotted box.

Although we could now code up the FORTRAN program easily enough, one requirement has been omitted so far: there is no *debugging output.* It is unusual for a computer program to work correctly the first time it is put on the machine, indeed, this hardly ever happens. Some of the errors, or "bugs", are simple errors of FORTRAN

grammar; these are detected by the FORTRAN compiler and give rise to diagnostic messages. But these are by no means the only possible errors. A FORTRAN program which compiles without diagnostics speaks grammatically correct FORTRAN. But people can speak grammatically correct English, and still say utter and complete nonsense! The same is true of programs which "speak" grammatically correct FORTRAN. Suppose that you put a minus sign at a point where your calculation requires a plus sign. There is no way in which the *compiler* can see that you have made a mistake. The compiler is a computer program for translating from FORTRAN into machine language. The compiler can not, and will not, read your mind!

Thus you, not the compiler, must "debug" your program. All programs must be written from the start in such a way that the inevitable "bugs" can be located by study of the printed output. A program containing a loop, as this one does, can not be debugged by looking merely at the final answer. Some nonsense is likely to come out (if the program does not "bomb out" before reaching the stage at which the final answer is to be printed, in which case nothing at all comes out!). Then you have no way of telling just how the program produced this particular nonsense. Thus, you must obtain output each time the loop is traversed, enough output so that you can follow all that happens in its full, horrible detail.

Therefore we shall include a WRITE statement within the loop, immediately ahead of the termination test (this is usually the best place for debugging output). We shall write out everything likely to be useful, such as the values of P, Q, RATIO, and TOLER (if in doubt, it is better to output too much than too little). We shall also precede, and follow, this WRITE statement with comment cards indicating that this is debugging output, which can be removed from the program after the program is working properly.

```
    1 FORMAT (7E16.6)
      A=9.6
      TOLER=0.000001
      P=3.0
C     LOOP STARTS HERE
 1500 Q=0.5*(P+A/P)
      RATIO=(P-Q)/Q
C***  DEBUGGING OUTPUT
      WRITE (3,1) P,Q,RATIO,TOLER
C***  END OF DEBUGGING OUTPUT
      IF(RATIO**2-TOLER**2) 2000,2000,2500
 2000 P=Q
      GO TO 1500
C     LOOP HAS ENDED
 2500 WRITE (3,1) A,Q
      STOP
      END
```

All right, this is the program. Now let us look at the output numbers from an actual machine run with this program:

```
 0.300000E 01     0.310000E 01     -0.322581E-01     0.100000E-05
 0.960000E 01     0.310000E 01
```

Clearly, something is very wrong. The last line should contain the correct answer, which we know to be 3.09839 (to six figures), whereas in fact it contains an incorrect answer, 3.10000.

Here is where the debugging output proves its worth! The first line is debugging output, and contains values of P, Q, RATIO, and TOLER, in that order. Let us check these! P=3.0 is correct, and so is Q=3.1; with these, the ratio (P-Q)/Q is indeed equal to -0.0322581; and TOLER is indeed as printed. So everything is correct -- except the answer! What on earth has gone wrong?

A clue to the riddle is provided by the fact that there is only one line of debugging output, indicating that we have reached this point in the program only once. But this means that we must have left the loop immediately after the first computation of Q, whereas we should have gone through the loop once more, to compute an improved value of Q. We conclude that something is wrong with the termination test.

Let us therefore compare the coding of the IF statement in the FORTRAN program, with the termination test shown in the flow diagram on page 29. Sure enough, there is that evil bug: the statement numbers in the FORTRAN code have been written in opposite sequence to what is on the flow diagram, i.e., the FORTRAN statement should have read "IF(RATIO**2-TOLER**2) 2500,2000,2000", rather than our IF statement on page 30. Both are quite legal FORTRAN, but only one of them is right for our program.

The corrected program reads:

```
    1 FORMAT (7E16.6)
      A=9.6
      TOLER=0.000001
      P=3.0
C     LOOP STARTS HERE
 1500 Q=0.5*(P+A/P)
      RATIO=(P-Q)/Q
C***  DEBUGGING OUTPUT
      WRITE (3,1) P,Q,RATIO,TOLER
C***  END OF DEBUGGING OUTPUT
      IF(RATIO**2-TOLER**2) 2500,2000,2000
 2000 P=Q
      GO TO 1500
C     LOOP HAS ENDED
 2500 WRITE (3,1) A,Q
      STOP
      END
```

Well, let us submit this corrected program, and look at the output. (*)

```
 0.300000E 01    0.310000E 01    -0.322581E-01    0.100000E-05
 0.310000E 01    0.309839E 01     0.520562E-03    0.100000E-05
 0.309839E 01    0.309839E 01     0.135493E-06    0.100000E-05
 0.960000E 01    0.309839E 01
```

A heart-felt sigh of relief is in order. The final value, 3.09839, agrees to six significant digits (all that is printed out here) with both the hand-computed answer and with the result we obtained by looking at a table of square roots. Furthermore, a full study of the debugging output shows that we have gone through the steps of the loop three times, each time doing the correct thing (you should check this!).

(*) These numbers were produced on a CDC 6400 computer. On another machine, the numbers in the third column of the table may turn out to be different. See page 60.

Section E: Making the Program More Efficient.

Although the program now works, in the sense of giving the right answer at the end, only a beginning programmer would stop at this point. An experienced programmer would now take a second look at the program, to see whether machine time and/or machine memory space can perhaps be saved by altering the code slightly, without altering the answer. This is called "making the program more efficient". It can be very important for production programs taking a long time on the machine.

First of all, we notice that our termination test (the IF statement) implies a computation of TOLER**2 every time we go around the loop. Since the quantity TOLER itself does not alter, it would save machine time to compute TOLER**2 once and for all, before entering the loop. Thus, we may precede statement 1500 by "TEST=TOLER**2", and we may then replace the IF statement by "IF(RATIO**2-TEST) 2500,2000,2000". This gives exactly the same printed output numbers, in less machine time.

Next, when we look at the printed output we see that we have traversed the loop three times altogether (there are three lines of debugging output). The third time was quite unnecessary, since the answer did not change (to six place accuracy) as a result of the third iteration.

The reason for this is that our termination test has been too severe. We have demanded that the estimates P and Q should agree with each other to six significant figures. But the iteration scheme which we are using is so efficient that an input value P which is correct to only *three* significant figures is good enough to make the improved value Q correct to *six* significant figures. (Those of you who are mathematically inclined may care to provide a proof of this). Thus, we are allowed to replace the statement setting the quantity "TEST" by the less stringent value: "TEST=TOLER". With this change, the output from the program differs from the output shown on page 31, by the absence of the third line (the one starting with 0.309839E 01); the final answer is the same. (Differences in the final results may occur, however, if the value of TOLER is so small that we get into trouble due to the limited precision with which numbers are stored in the machine; see Chapter VI).

With all these changes, and with omission of the debugging output, the revised program reads:

```
    1 FORMAT (7E16.6)
      A=9.6
      TOLER=0.000001
      P=3.0
      TEST=TOLER
C     LOOP STARTS HERE
 1500 Q=0.5*(P+A/P)
      RATIO=(P-Q)/Q
      IF(RATIO**2-TEST) 2500,2000,2000
 2000 P=Q
      GO TO 1500
C     LOOP HAS ENDED
 2500 WRITE (3,1) A,Q
      STOP
      END
```

Summary and Formal Definitions for Chapter III.

1. *Statement Numbers:*

   Unsigned integers between 1 and 9999, inclusive, located in columns one to five of the card. No two statements can have the same statement number, but otherwise statement numbers can be assigned at will, in any sequence. Statement numbers *must* be assigned to: (1) statements to which control is transferred, (2) statements immediately following a GO TO or an IF statement, and (3) FORMAT statements.

2. *Executable and Non-executable Statements:*

   An "executable statement" is one which is translated into a command to the machine to carry out some operation during the execution phase. A "non-executable statement" is one which tells the FORTRAN compiler how to translate some other statement (or statements), but which

is not itself translated into machine language, or obeyed during the execution phase. Executable statements encountered so far are: arithmetic assignment statement, GO TO, IF, WRITE, and STOP. Non-executable statements encountered so far are: END and FORMAT. The END statement instructs the compiler to cease translation of this program. The FORMAT statement instructs the compiler exactly how to translate some WRITE statement. For a complete list of executable and non-executable statements, see page 151.

3. *Unconditional Transfer of Control:*

   GO TO *n* where *n* is the statement number of an executable statement in the program. Statement number *n* is obeyed next.

4. *Conditional Transfer of Control:*

   IF(*W*) *i,j,k* where *W* is an arithmetic expression, *i, j, k* are statement numbers of executable statements in the program.

   Control is transferred to:
   statement number *i* if the value of *W* is negative,
   statement number *j* if the value of *W* is zero,
   statement number *k* if the value of *W* is positive.

5. *Termination of Execution:*

   STOP When this is encountered during the execution phase, program execution stops.

   Important Note:
   In some installations, this statement causes complete stopping of the machine, rather than automatic reading in of the next job in the batch of jobs. In those installations, the statement "STOP" should be replaced by: "CALL EXIT".

DRILL EXERCISES FOR CHAPTER III

1. Which of the following statement numbers are valid?

   Card Column: 123456789...

   (a) 4798PSS=1.0
   (b) 798 PSS=1.0
   (c) 3205PSS=1.0
   (d) 352467PSS=1.0
   (e) 1) PSS=1.0
   (f) 12P PSS=1.0
   (g) 36669PSS=1.0

2. Which of the following statements are invalid, and why?

```
   IF(6.6) 48,58,68
   IF(Y-6.6) 48,58,68
   IF(Y-6) 48,58,68
48 GO TO 677655
58 GO TO 58
68 GO TO STOP
```

3. Write flow diagrams and short program segments to do the following:

   (a) If F is positive, move the larger one of C and D to location U; if F is negative, move the smaller one of C and D to location U. Assume that F is not zero.

   (b) If both A and B lie between 2.5 and 3.5 (exclusive of the endpoints of that interval), move D to location TEST. If A lies inside the interval, B outside, move C to location TEST. If A lies outside the interval, B inside, move D to TEST. If both A and B lie outside the interval, move C to location TEST.

4. What is wrong with the following program segments?

(a)
```
   B=1.35
 5 IF(B-13.5) 6,8,6
```

(continued on the next page)

```
    6 B=B+1.35                    (continued from preceding page)
      GO TO 5
    8 STOP
```

(b)
```
      A=A+1.36
      GO TO 2500
      B=A**2
2500  C=A+B
```

(c)
```
      A=B
2300  IF(A-B) 2400,2300,2500
2400  A=A+1.0
2500  B=B+2.0
```

5. Construct IF statements which test the truth of each assertion below, and which transfer control to statement number 2000 if the answer is "yes", to statement 3000 if the answer is "no".
   (a) A is greater than twice B,
   (b) A is greater than or equal to twice B,
   (c) A is unequal to twice B,
   (d) A is less than twice B,
   (e) A is less than or equal to twice B,
   (f) A lies within the interval from twice B to three times B (excluding the endpoints of the interval),
   (g) same as (f), but including the endpoints of the interval,
   (h) X is greater than all of A, B, and C,
   (i) X is greater than at least one of A, B, or C,
   (j) X lies between 1.6 and 2.4, and Y is either less than 3.7 or greater than 4.9,
   (k) X does not lie between 1.6 and 6.9, and neither does Y,
   (l) X equals A and Y equals B.

6. State in words the condition under which control is transferred to statement number 1500 in each of the following IF statements:
   (a) IF(A**2-9.61) 1500,2000,2000
   (b) IF(A**2-9.61) 1500,1500,2000
   (c) IF(A**2-9.61) 2000,1500,2000
   (d) IF(A**2-9.61) 2000,2000,1500
   (e) IF(9.61-A**2) 2000,2000,1500
   (f) IF((A-1.0)*(3.7-A)) 2000,2000,1500

PROGRAMMING EXERCISES FOR CHAPTER III.

1. Revise the program on page 25 for an initial debt of 280.00, with a monthly repayment of 35.84.
2. Same as (1), but interest is charged at 1.2 percent of the unpaid balance each month.
3. Alter the square root program (page 31) to start from an initial guess P=1.0, which is much inferior to the starting guess we used there. How many steps are required for the same final accuracy?
4. Alter the square root program (page 31) to stop when an accuracy of four (rather than six) significant figures is reached. Compare your answers with the six-figure output on page 31.
5. Use the following iteration scheme for finding the cube root of a given number "a".

   $$q = \frac{1}{3}(2p + \frac{a}{p^2})$$

   Write a program to evaluate the cube root of 9.6 to six significant places.

CHAPTER IV

INPUT AND OUTPUT

Section A: Introduction.

So far, the numbers with which our FORTRAN programs have done their calculations have been written directly into the programs. For example, when we wanted to take the square root of 9.6, the FORTRAN statement "A=9.6" appeared within the program. Similarly, in the debt repayment program, the initial amount of the debt was set by the statement "DEBT=300.00"; and so on.

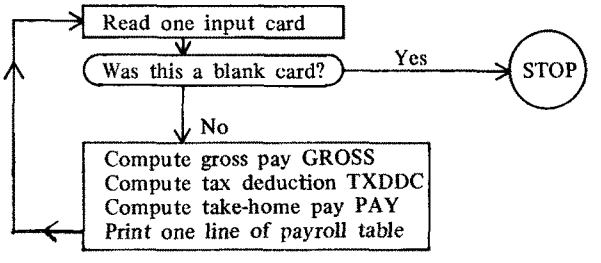
Such programs suffer from a lack of flexibility. If we are interested in the square root of some other number, or in the repayment of some other debt, then the program itself must be revised.

Furthermore, there are important applications, particularly in the area of business data processing, where programs of this type are entirely unsuitable. Let us consider the following payroll problem:

A company wants to make out a payroll for one week. For each employee, there is an input card on which has been punched: the hourly wage rate paid to the man, and the number of hours he worked during the week. Twelve percent of his gross pay is to be withheld as a taxation deduction. The printed output is to show the input data for each employee, and next to them, the computed values of the gross pay, taxation deduction, and take-home pay.

Clearly, it would be out of the question to write a new FORTRAN program for each such calculation, one program for every employee! Rather, we want a program which tells the computer to *read input cards during the execution phase,* and which processes the input data on each card; card after card is to be read and processed, until the last card is encountered. Only then is the job finished. It is typical of this, as of many other, business type data processing problems, that the amount of actual computation done by the machine on any one set of data is quite minor. The main time and effort is spent on input of data, and output of results.

In our payroll example, we shall use a blank card (no data punched on it) to indicate that all the employee data cards have been processed. The computer keeps reading employee data cards (pay rates and hours worked), until it encounters a blank card, at which point it stops. The flow diagram follows:



Section B: Reading Input Data.

Input data numbers are read from input data cards, and are transferred to named memory locations inside the machine. The memory locations are specified by means of an *input list,* which is a *simple list.*

A *simple list* consists of identifier names, separated from each other by commas. There is no comma at the beginning of such a list, and no comma at the end of the list.

Example:

The following is a simple list, consisting of six identifier names:

```
OH,WHAT,AN,AWFUL,HEAD,ACHE
```

Note that only identifier names are permitted in such a list, not specific numbers (FORTRAN constants) and not compound arithmetic expressions. Thus "21.35" is not a valid list element, and neither is "ACHE**2" or "HEAD+ACHE".

To keep things simple, we shall arrange it so that any one data card can have one, two, three, or four numbers on it, but no more than four. Thus, if we want to read six numbers into the memory locations specified on the list above, we shall read the first four numbers from one data card, and the remaining two numbers from the next data card. The FORTRAN statements required for doing this are:

```
2 FORMAT (4F10.0)
  READ (1,2) OH,WHAT,AN,AWFUL,HEAD,ACHE
```

Explanation: The FORMAT statement has statement number "2" (we could have used any number had we so wished). This FORMAT statement specifies, in a way which will be explained in Chapter VII, that up to four data numbers can appear on one data card, and specifies just how these numbers must appear on the card (see below).

The READ statement instructs the machine to read data cards at execution time. Since there are six memory locations named in the list, six numbers are read from data cards (four from the first data card, two more from the card after it). The first number read is stored into memory location "OH", the second number is stored into memory location "WHAT", and so on, till the sixth number is stored into location "ACHE".

The two integers appearing inside the parentheses of the READ statement have the following meaning: "1" is the number assigned to the card reader; and "2" is the statement number which we have given to the FORMAT statement.

The numbers assigned to the card reader and to the line printer are unfortunately not the same in all installations. So far we have used "3" for the number assigned to the line printer, and "1" for the number assigned to the card reader. Though a fairly common choice, this is by no means a universal rule at all installations. In order to make our programs acceptable at all installations, we shall use the identifier name "NREAD" for the number assigned to the card reader, and we shall use the identifier name "NPRNT" for the number assigned to the line printer. Note that both names start with the letter "N", and are therefore appropriate names for integers. For the time being, our programs will start in the following standard fashion:

```
1 FORMAT (7E16.6)
2 FORMAT (4F10.0)
  NREAD=1
  NPRNT=3
```

The numbers "1" and "3" on the last two lines may have to be replaced by other numbers for your installation (you must ask what they are). But no other changes will be needed in our programs. The first FORMAT statement, the one with statement number equal to 1, will be used for print-out, with WRITE statements of the form:

WRITE (NPRNT,1) *List*

where *List* is a simple list. The second FORMAT statement, the one with statement number equal to 2, will be used for reading input data, with READ statements of the form:

READ (NREAD,2) *List*

where *List* is again a simple list. The list in a WRITE statement is called an *output list,* the list in a READ statement is called an *input list.*

We know, already the form in which numbers are printed out by such a WRITE statement with this particular FORMAT statement, see page 8.

Let us now discuss how *input cards* must be prepared to be acceptable with our standard READ statement. Only the first forty columns of the card are used, and these are divided into four equal "fields":

First field: Columns 1 to 10

Second field: Columns 11 to 20

Third field: Columns 21 to 30

Fourth field: Columns 31 to 40

If a data number is zero, the corresponding field may be left blank. Otherwise, the desired number must appear entirely within its own field (it must not spill over into neighbouring fields), and it must appear as an ordinary decimal number *with a decimal point.* As long as these simple rules are obeyed, it does not matter just where the number appears within its field. But it is good data preparation practice to keep away from the boundaries of the field, i.e., to put each number roughly into the middle of its field.

If less than four numbers are required on a data card, no further numbers should be put on that card, since the machine would not pay any attention to them, anyway. For example, the READ statement:

READ (NREAD,2) OH,WHAT,AN,AWFUL,HEAD,ACHE

requires *two* data cards; the second of these cards should contain only two numbers (the desired values of HEAD and ACHE), in the first two fields of that card. The last two fields should be left blank.

Beginning programmers tend to become confused about *when* input occurs as a result of a READ statement in the source language program, and *when* output occurs as a result of a WRITE statement in the source language program. Referring back to Chapter I, page 2, we recall that there are two distinct phases in the handling of a job:

1. The *compile phase,* during which the FORTRAN compiler is in charge and translates the FORTRAN program into machine language; and
2. The *execution phase,* during which the translated (object) program is in charge, with a master (supervisor) program merely exercising more or less paternal supervision.

A READ (or WRITE) statement in the FORTRAN program causes input (or output)

in phase 2 above, the execution phase, *not* in phase 1. In fact, it is not the FORTRAN statement READ (or WRITE), but its *translated version,* which causes the input (or output).

All FORTRAN program cards are read and translated, before the first data card is demanded from the card reader (which happens only after the execution phase has been reached). Similarly, the full listing of all the FORTRAN program cards appears on the printed output, before anything at all is printed by any WRITE statement.

The FORTRAN program cards are read and listed onto the output page under control of the compiler. Only after the compiler has relinquished control, and the execution phase has commenced, only then are data cards read (with translated READ statements) and output numbers printed (with translated WRITE statements).

We recall from Chapter I, page 3, that there is a control card which signals the end of the compile phase. All FORTRAN program cards must be *ahead* of this control card in the job deck. All data cards must be *after* this control card in the job deck.

Section C: A Payroll Problem.

Let us now code up the payroll problem on page 35, using the flow diagram there. Before we can do this, however, we must still decide how we are going to recognize the blank card which signals the end of the batch of input data cards.

With our READ command, a blank field on an input card is interpreted as the number zero. According to the statement of the problem, the first number on each input card is the hourly wage rate paid to the man. Since an hourly wage rate is never zero, we can recognize a blank card by testing what value of WAGE has been read in. A zero value means we have come to the blank card. The program is:

```
    1 FORMAT (7E16.6)
    2 FORMAT (4F10.0)
      NREAD=1
      NPRNT=3
C     CHECK THESE NUMBERS FOR YOUR INSTALLATION.
C     NOW START THE LOOP
 1500 READ (NREAD,2) WAGE,HOURS
      IF(WAGE) 3000,3000,2000
 2000 GROSS=WAGE*HOURS
      TXDDC=0.12*GROSS
C     THIS IS A 12 PERCENT TAX DEDUCTION
      PAY=GROSS-TXDDC
      WRITE (NPRNT,1) WAGE,HOURS,GROSS,TXDDC,PAY
      GO TO 1500
C     WE REACH THE STATEMENT BELOW AFTER READING A BLANK CARD
 3000 STOP
      END
```

This is the FORTRAN program. The job deck is described in detail in Chapter XV. Briefly, it consists of a job card (and perhaps other control cards), followed by FORTRAN program cards (that is, the program appearing above). Then comes the control card that signals the end of the compile phase. Then come the data cards, each with two numbers on it (the values of WAGE and HOURS, respectively), in the first two fields. The last

meaningful data card is followed by a blank data card, to indicate the "end-of-data". In some (but not in all) installations, an "end-of-job" control card is required at the very end of the job deck.

Let us suppose that there are just three employees. The first gets $2.25 per hour, and has worked 35 hours. The second gets $2.75 per hour, and has worked 26 hours. The third gets $3.20 per hour, and was sick that week. The input data cards must be:

Card No:	*First Field (Cols. 1-10)*	*Second Field (Cols. 11-20)*	*Third Field (Cols. 21-30)*	*Fourth Field (Cols. 31-40)*
1	2.25	35.0	blank	blank
2	2.75	26.0	blank	blank
3	3.20	0.0 (or blank)	blank	blank
4	blank	blank	blank	blank

Please notice that the input number 35 has been written *with a decimal point,* that is, as "35.0", and the input number 26 has been written as "26.0". *This is a must.*

Assuming that these input cards have been prepared correctly, here is the numerical output produced by this program:

0.225000E 01	0.350000E 02	0.787500E 02	0.945000E 01	0.693000E 02
0.275000E 01	0.260000E 02	0.715000E 02	0.858000E 01	0.629200E 02
0.320000E 01	0.000000E 00	0.000000E 00	0.000000E 00	0.000000E 00

The first two columns of this table merely repeat the values of WAGE and HOURS on the corresponding input card. This is called *"echo-checking the input data",* and is absolutely essential. (The idea of the term "echo-check" is that you "shout" the numbers into the machine, and get a printed "echo" back). Without echo-checking, you have no idea what numbers the machine is working on. You may think you know, merely by looking at the data cards. But you do not know! The data cards may have been punched incorrectly, or they may be out of order in the card deck. By looking at the data cards, you can tell what numbers you *wanted* the machine to read. But *only the echo-check tells you what numbers the machine did actually read.*

We note that this output is in a pretty awkward form for a payroll. Decimal points must be shifted all over the place, and fractions of cents are printed out. This is due to our choice of output format (that is, the FORMAT statement with statement number 1), which is a good general-purpose choice, but is not particularly suitable for this specific application. Better choices of output format will be presented in Chapter VII.

Section D: Tests on Input Numbers. Messages in the Output.

Our payroll program works if the input numbers are correct. But there is no test to see whether the input numbers are in fact correct, or even reasonable. This is a serious omission. *In all data processing, it must be assumed that some of the input cards will be prepared incorrectly.* Mistakes will happen, and precautions must be taken against them.

Guarding against mistakes in the data, in such a way that no mistake is missed, but yet the least amount of computer time is wasted, that is the real art of data processing! This is what separates the sheep from the goats in programming.

For example, it may be known that the hourly wage rates in this particular firm

range from $1.50 per hour to $4.25 per hour. We should then test the value of WAGE read in from the data card, to make sure it falls within this range. Similarly, the number of HOURS worked can not be negative, nor should it be more than some limit (say, 40 hours per week).

Suppose we do detect an unreasonable value of WAGE or HOURS, or both, on a data card. What then? How do we make the machine tell us what happened?

FORTRAN allows us to print *messages on the output page.* For example, the message could be a line reading:

***** UNREASONABLE VALUE OF WAGE ON LINE BELOW

This warns the reader that the next line of the payroll table must be treated with caution. The FORTRAN statements for printing out this particular message are:

```
      WRITE (NPRNT,1810)
 1810 FORMAT (47H ***** UNREASONABLE VALUE OF WAGE ON LINE BELOW)
```

Explanation:

The WRITE statement has no output list at all; thus no numbers are printed out. The "1810" within the parentheses of the WRITE statement directs us to FORMAT statement number 1810 (we could have chosen any number, of course). We have written that FORMAT statement immediately afterwards, but it could appear anywhere in the program.

The message to be printed out is contained within the parentheses of the FORMAT statement. The symbols "47H" indicate that the *next 47 characters* represent a message of length 47; *this count includes blank characters,* e.g., the blank character between "OF" and "WAGE" is counted as one character. The message itself, as given before, contains 46 characters, not 47. But we have deliberately added a blank character at the very beginning, because the FORTRAN system automatically omits the first character of such a message, in printed output. (*) That is, the blank character immediately following the "47H" is not printed out at all. The printed line starts with an asterisk "*" in the *first* position.

The count of the number of characters in a message is often wrong, in practice, thereby leading to trouble. We recommend the following *safety precautions:*

1. After counting the number of characters in a message, and adding 1 to this to allow for the omitted blank character at the beginning, specify still one more as the number *n* in the *n*H specification. E.g., our message above has 46 characters, the blank character in front makes it 47, and we now advise you to specify 48H rather than 47H.
2. To make up for this, leave three blank spaces between the end of the actual message, and the closing bracket of the FORMAT statement.

With these precautions, the revised FORMAT statement reads:

```
 1810 FORMAT (48H ***** UNREASONABLE VALUE OF WAGE ON LINE BELOW   )
```

One of the three blanks at the end, between "BELOW" and the closing bracket, is included

(*) In full FORTRAN IV, as opposed to Basic FORTRAN IV, the first character is treated as a coded command to the line printer, not as a symbol to be printed. For example, the line printer can be instructed to skip to the start of a new page, before printing this line. In Basic FORTRAN, the first character is simply ignored altogether.

as part of the message as a result of the 48H specification. The remaining two blanks are ignored by the FORTRAN compiler (blanks are counted within a declared message, but only there; other blanks are ignored). The effect of these precautions is that nothing serious happens if we have miscounted by one unit, in either direction.

Messages in printed output can be used, not only for signalling failure conditions, but also for *table headings.* In the payroll program, it would be nice to have proper headings at the top of each column of numbers, particularly so if the table is to be read by someone other than yourself (for example, your boss). To help in this, FORTRAN allows two further specifications:

1. The specification 6X specifies 6 *blank spaces* on the printed output.

2. The slash "/" specifies the *end of one line,* and the start of the next line.

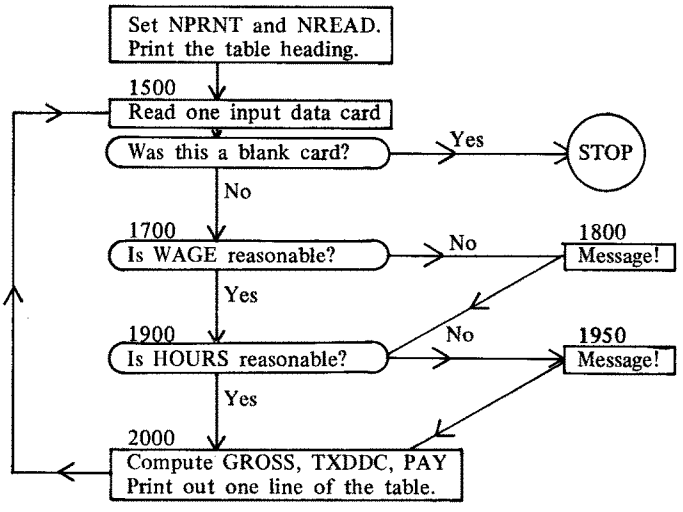
Using these two features, let us write down the FORTRAN statements for producing a table heading for our payroll:

```
      WRITE (NPRNT,1100)
 1100 FORMAT (6X,9HWAGE RATE,9X,5HHOURS,11X,5HGROSS,12X,3HTAX,13X,3HPAY,
     1/)
```

This specifies five spaces at the beginning of the line (five, not six, because the first character is omitted), followed by the nine-character message "WAGE RATE", followed by nine spaces, followed by the five-character message "HOURS", and so on. The last thing specified is a slash, which has the effect of leaving a blank line below the table heading, to separate the table heading from the body of the table. We must use a continuation card to complete the FORMAT statement, since the comma after "3HPAY" is already in column 72 of that card, and column 72 is the last column used by FORTRAN. The spaces between the messages in this FORMAT statement have been arranged so that each column heading appears centered on the appropriate column. The output from this program will be shown in full, a bit later.

Note that the different specifications in this FORMAT statement are separated from each other by commas. This is necessary.

We are now ready to recode the payroll program. First, of course, comes the revised flow diagram:



The FORTRAN program reads:

```
C       PAYROLL PROGRAM
      1 FORMAT (7E16.6)
      2 FORMAT (4F10.0)
        NREAD=1
        NPRNT=3
C       CHECK THESE NUMBERS FOR YOUR INSTALLATION
C       NOW COMES THE TABLE HEADING
        WRITE (NPRNT,1100)
 1100 FORMAT (6X,9HWAGE RATE,9X,5HHOURS,11X,5XGROSS,12X,3HTAX,13X,3HPAY,
     1/)
C       LOOP STARTS HERE
 1500 READ (NREAD,2) WAGE,HOURS
        IF(WAGE) 3000,3000,1700
C       NOW TEST VALUES OF WAGE AND HOURS
 1700 IF((WAGE-1.50)*(4.25-WAGE)) 1800,1900,1900
C       WAGE IS EITHER BELOW 1.50 OR ABOVE 4.25 ... SCREAM
 1800 WRITE (NPRNT,1810)
 1810 FORMAT (48H ***** UNREASONABLE VALUE OF WAGE ON LINE BELOW    )
C       NOW TEST HOURS
 1900 IF(HOURS*(40.0-HOURS)) 1950,2000,2000
C       HOURS IS EITHER NEGATIVE OR ABOVE 40.0 ... SCREAM
 1950 WRITE (NPRNT,1960)
 1960 FORMAT (49H ***** UNREASONABLE VALUE OF HOURS ON LINE BELOW    )
C       NORMAL COMPUTATION
 2000 GROSS=WAGE*HOURS
        TXDDC=0.12*GROSS
C       THIS IS A 12 PERCENT TAX DEDUCTION
        PAY=GROSS-TXDDC
        WRITE (NPRNT,1) WAGE,HOURS,GROSS,TXDDC,PAY
        GO TO 1500
C       WE REACH THE STATEMENT BELOW AFTER READING A BLANK CARD
 3000 STOP
        END
```

If you are puzzled by the IF statements with statement numbers 1700 and 1900, please re-read the discussion on page 27.

In order to debug this program, it is necessary to supply not only "good" data cards, but also data cards which are deliberately "bad", so as to test the failure printouts. Hence we supply two additional data cards, one with an unreasonable value for WAGE, the other with an unreasonable value for HOURS. The input data cards are:

Card No:	*First Field (Cols. 1-10)*	*Second Field (Cols. 11-20)*	*Third Field (Cols. 21-30)*	*Fourth Field (Cols. 31-40)*
1	2.25	35.0	blank	blank
2	2.75	26.0	blank	blank
3	3.20	0.0 (or blank)	blank	blank
4	9.99	34.0	blank	blank
5	2.25	-7.5	blank	blank
6	blank	blank	blank	blank

This time, we show an example of *a complete output* (not merely the output numbers, but also the listing of the control cards, FORTRAN program, etc.) from a particular FORTRAN IV compiler (namely WATFOR). We show the output first, and discuss it afterwards.

```
      $JOB              10.001          BLATT                    88885555   BLATT
      C     PAYROLL PROGRAM
 1        1 FORMAT (7E16.6)
 2        2 FORMAT (4F10.0)
 3          NREAD=1
 4          NPRNT=3
      C     CHECK THESE NUMBERS FOR YOUR INSTALLATION
      C     NOW COMES THE TABLE HEADING
 5          WRITE (NPRNT,1100)
 6     1100 FORMAT (6X,9HWAGE RATE,9X,5HHOURS,11X,5HGROSS,12X,3HTAX,13X,3HPAY,
           1/)
      C     LOOP STARTS HERE
 7     1500 READ (NREAD,2) WAGE,HOURS
 8          IF(WAGE) 3000,3000,1700
      C     NOW TEST VALUES OF WAGE AND HOURS
 9     1700 IF((WAGE-1.50)*(4.25-WAGE)) 1800,1900,1900
      C     WAGE IS EITHER BELOW 1.50 OR ABOVE 4.25 .. SCREAM
10     1800 WRITE (NPRNT,1810)
11     1810 FORMAT (48H ***** UNREASONABLE VALUE OF WAGE ON LINE BELOW    )
      C     NOW TEST HOURS
12     1900 IF(HOURS*(40.0-HOURS)) 1950,2000,2000
      C     HOURS IS EITHER NEGATIVE OR ABOVE 40.0 .. SCREAM
13     1950 WRITE (NPRNT,1960)
14     1960 FORMAT (49H ***** UNREASONABLE VALUE OF HOURS ON LINE BELOW    )
      C     NORMAL COMPUTATION
15     2000 GROSS=WAGE*HOURS
16          TXDDC=0.12*GROSS
      C     THIS IS A 12 PERCENT TAX DEDUCTION
17          PAY=GROSS-TXDDC
18          WRITE (NPRNT,1) WAGE,HOURS,GROSS,TXDDC,PAY
19          GO TO 1500
      C     WE REACH THE STATEMENT BELOW AFTER READING A BLANK CARD
20     3000 STOP
21          END
      $ENTRY
 WAGE RATE       HOURS          GROSS          TAX            PAY

  0.225000E 01    0.350000E 02    0.787500E 02    0.945000E 01    0.693000E 02
  0.275000E 01    0.260000E 02    0.715000E 02    0.858000E 01    0.629200E 02
  0.320000E 01    0.000000E 00    0.000000E 00    0.000000E 00    0.000000E 00
***** UNREASONABLE VALUE OF WAGE ON LINE BELOW
  0.999000E 01    0.340000E 02    0.339660E 03    0.407592E 02    0.298901E 03
***** UNREASONABLE VALUE OF HOURS ON LINE BELOW
  0.225000E 01   -0.750000E 01   -0.168750E 02   -0.202500E 01   -0.148500E 02

COMPILE TIME=  1.39 SEC, EXECUTION TIME=  0.31 SEC, OBJECT CODE=  1056 BYTES
ARRAY AREA=  0 BYTES, UNUSED=  13944 BYTES
```

The details of your own print-out are going to differ from this (unless your installation is also using WATFOR), but the general principles are the same. The output contains:

1. Listing of the job card, and other initial control cards (if any). For the form of your own job card, see Chapter XV.
2. Listing of the FORTRAN program, with non-comment statements numbered sequentially for easier reference. These sequential numbers have nothing to do with the statement numbers, nor do they influence the flow of control in the program. (Some compilers do not produce sequential numbers).
3. Listing of the control card (or cards) signalling the end of the compile phase (see Chapter XV). In the present instance, this control card reads $ENTRY.
4. The printed output generated by the object program, as a result of execution of the (translated) WRITE statements reached during program execution. This is the output you really wanted.
5. Some kind of termination message (see Chapter XVI).

Looking at the output on the preceding page, we observe that the program appears to be working properly. The table heading is right, the numbers are right, and failure messages are indeed printed out before the last two lines of the table.

Summary and Formal Definitions for Chapter IV.

1. Simple List:

A simple list consists of identifier names, separated by commas. There is no comma before the list, or at the end of the list.

2. Unit Numbers:

In FORTRAN, the card reader and the line printer attached to the computer are assigned "unit numbers", which are integers. These differ from installation to installation. We use the names NREAD and NPRNT for the unit numbers of the reader and the printer, respectively. These are set by assignment statements at the beginning of the FORTRAN program.

3. Reading Data Cards:

For the time being, each data card has four "fields". Field 1 consists of columns 1-10, field 2 of columns 11-20, field 3 of columns 21-30, field 4 of columns 31-40. Each field must be either blank or contain a decimal number with an explicit decimal point. The standard FORMAT statement for reading data cards is:

```
2 FORMAT (4F10.0)
```

The FORTRAN command for reading data cards is:

READ (NREAD,2) *List*

where *List* is a simple list. The READ command initiates reading of a new data card. Numbers are read off the data cards, one by one, and stored into the memory locations appearing in the list. Exactly enough numbers are read to exhaust the list, four from the first data card, four more from the next data card, and so on, until the list is completed. The data cards themselves appear in the job deck after the control card which signals the start of the execution phase.

4. Printing a Line of Numbers:

```
1 FORMAT (7E16.6)          is the standard FORMAT statement for now.
WRITE (NPRNT,1) List       is the standard WRITE statement for now.
```

The WRITE statement initiates writing of a new line of output. The contents of the memory locations appearing in the list are printed out, one by one, seven to a line of print, in E-format. For explanation of this format, see page 8.

5. *Printing Message Lines:*

The field separators are the comma "," and the slash "/". The slash indicates the end of a line, and start of a new line.

A Hollerith field descriptor has the form *n*H followed by exactly *n* characters. These *n* characters (blanks are counted!) appear on the output (except for the very first character in each line, which is suppressed).

A blank field descriptor has the form *n*X and gives rise to *n* adjacent blank characters on the output line, i.e., to *n* blank spaces (unless this blank field is the first field on a line, in which case the first of the *n* blanks is suppressed, and only *n*-1 blanks appear).

The FORMAT statement must have a statement number. The word FORMAT appears starting in column 7, then a left bracket, then format information, then a closing right bracket. The format information is in the following sequence:

1. (Optional) Any number of slashes
2. A field descriptor
3. A field separator
4. A field descriptor
5. Another field separator

..... (and so on)

Last. (Optional) Any number of slashes

The WRITE statement for printing message lines is:

WRITE (NPRNT,*m*)

where *m* is the statement number assigned to the FORMAT statement containing the message to be printed. The format information in the FORMAT statement is examined reading from left to right. Each field descriptor gives rise to a field of adjacent characters on the printed line; each slash starts a new line. The first character of the first field of each new line is ignored; it is not printed out by the printer.

DRILL EXERCISES FOR CHAPTER IV.

1. Explain the difference between data appearing in the FORTRAN program, and data read in at execution time. Explain why a program which does not read data at execution time can be run on the computer only once.
2. Explain what would be wrong with placing input data cards for a program immediately after the READ statement which is meant to read those data cards. Where in the job deck do the data cards belong?
3. What is the reason for using identifier names NREAD and NPRNT for the numbers assigned to the card reader and line printer, respectively? Explain what would be wrong with placing the statement NPRNT=3 (or its equivalent for your installation) at the end of the FORTRAN program, rather than at the beginning of the program.
4. Consider the following statement:
   READ (NREAD,2) ALL,OF,US,ARE,VERY,CUTE,GIRLS
   How many data numbers are read in? How many data cards are required? For each card, state how many data numbers should appear on it, and where on the card they should appear. Also state the memory location into which each number is placed as a result of the READ statement.
5. Inspect the following statements for errors, and correct the errors (if any); assume our standard FORMAT statements are in the program.

   a) READ (NREAD,2 LIFE,IS,SHORT,ENOUGH
   b) READ (NREAD,FORMAT) LIFE,IS,SHORT

c) READ (NPRNT,2) WAGE,HOURS
d) WRITE (NPRNT,2) WAGE,HOURS
e) WRITE NPRNT,1 WAGE,HOURS
f) READ NREAD,2 WAGE,HOURS
g) READ (NREAD,2,WAGE,HOURS
h) READ (NREAD,2),WAGE, HOURS
i) READ (NREAD,2) WAGE,HOURS,

6. Inspect the following FORMAT statements for errors:

```
1800 FORMAT (25H ALL HELL HAS BROKEN LOOSE    )
1900 FORMAT (26H ALL HELL HAS BROKEN LOOSE
2000 FORMAT,(26H ALL HELL HAS BROKEN LOOSE)
2100 FORMAT (27H ALL HELL HAS BROKEN LOOSE)
     FORMAT (26H ALL HELL HAS BROKEN LOOSE)
```

7. Revise the payroll program so as to allow for a graduated income tax deduction, as follows: 12 percent is deducted for people with an hourly wage less than or equal to $2.30; 15 percent is deducted from people with an hourly wage between $2.31 and $3.45, inclusive; and 20 percent is deducted from people with an hourly wage greater than or equal to $3.46.

8. Revise the square root program of page 31 so that it reads input cards, each containing a number whose square root is to be computed. Use P=1.0 as the initial guess in each case. After a square root has been found, go back to read the next number whose square root is wanted. Continue until a blank card or a negative value of A is encountered.

9. Improve the program of exercise 8 as follows: (a) Supply a table heading for the table of numbers and their square roots, and (b) Test whether the input number is negative; if so, print out an error message, then skip over the computation of the square root in such a way that the line of the table below that error message contains the faulty (negative) number, followed by a value of zero for the "square root".

10. Revise the debt repayment program on page 25 so as to read input data cards, each card containing data numbers for DEBT and PAYMT. After computation for one case has finished, do not stop the machine, but rather go back to read the next set of data numbers. Stop only on a blank card.

11. Improve the program of exercise 10 as follows:
    a) Supply a table heading at the start of each new computation.
    b) Test whether DEBT is negative; print an error message, but do not compute anything, when this happens.
    c) Test PAYMT not only for a possible negative value, but also for the possibility that the value may be so small that the debt is not reduced at all, but continues to increase. If so, print an error message, but do not compute anything. Explain what would happen if you did proceed to compute in such a case.

PROGRAMMING EXERCISES FOR CHAPTER IV.

1. Write and debug an expanded version of the payroll program, which allows for the possibility of overtime hours, paid at the rate of 1½ times the normal wage rate for the man. Each input card now contains three numbers, namely WAGE, HOURS, and OVTIM. Make sure to include the value of OVTIM in the printout, revise the table heading accordingly; and test each input card for the following requirements: no overtime is allowed if straight time is less than 40 hours; and the value of OVTIM must not be negative, or greater than 15 hours. Print out appropriate error messages if either condition is violated. Supply data cards to test all failure cases.

2. Revise the debt repayment program of drill exercise 11 by allowing different interest rates, the value of RATE to appear as the third number on each input card. The value is to be a percentage rate on the unpaid balance each month, and the program itself should convert from a percentage to a fraction (e.g., from 0.8 percent to the fraction 0.008). Make sure that the echo check prints out that input number actually read in from the card, not its converted form. Otherwise it would not be an honest echo check. Test the input value of RATE, and print out a warning message if it is below 0.2 percent or above 3.5 percent.

CHAPTER V

INTERPRETATION OF DIAGNOSTIC MESSAGES AND EXECUTION TIME ERROR MESSAGES

Section A: Introduction.

When the FORTRAN compiler encounters a statement which is grammatically incorrect FORTRAN, it "flags" this statement on the printed output, with some sort of error message. These error messages are called *diagnostics,* since their purpose is to help you, the "doctor", to "diagnose" the illness of your program. These diagnostic messages differ from compiler to compiler, quite considerably. See Chapter XVI for the diagnostic messages from your own compiler.

Besides diagnostic messages during the compile phase, another kind of error message exists, namely *execution time error messages* which tell you that something has gone wrong during execution of the object (translated) program. For example, at some point during program execution you may have made an attempt to divide by zero. Execution time error messages are produced by the supervisor program which supervises execution of the object program. These messages also differ greatly from one supervisor to another, not only in the clarity of the message, but in the degree of checking which is done, and in the action taken when an error has been seen. For example, some supervisor systems detect an attempt to divide by zero, produce an error message, and stop further program execution at this point. Other supervisors may allow the program to continue executing further. Still other supervisors do not produce an error message automatically, but expect the programmer to request (by means of special orders in his program) information about whether such an error has occurred; nothing is printed out until such a request is encountered. See Chapter XVI for execution time error messages from your supervisor.

In spite of all these differences, a number of general principles apply to the interpretation of all error messages. These general principles form the subject of the present Chapter.

Section B: Interpretation of Diagnostics.

Let us look at a program which is (deliberately) chock full of errors. First we give the program itself:

```
C     SAMPLE PROGRAM WITH DIAGNOSTICS
      READ (1,1 A,B,C
      FORMAT(4F10 0)
      IF(N) 3,3
      GO TO 77
      A=A*B/C**2
      WRITE (3,77)
```

```
   77 FORMAT (17H DATA UNACCEPTABLE)
      STOP
/*
```

The card with the symbols "/*" in columns 1 and 2 is a normal control card for the end of the compile phase in Disk Operating System/360 on the IBM/360. We shall assume that the program has been submitted (with appropriate control cards, of course) to the FORTRAN compiler under that system. The printed output which results appears below, and its explanation follows afterwards.

```
            C     SAMPLE PROGRAM WITH DIAGNOSTICS
                  READ (1,1 A,B,C
                           $
01) SYNTAX
                  FORMAT(4F10 0)
                         $       $
01) LABEL                 02) SYNTAX
                  IF(N) 3,3
                         $
01) SYNTAX
                  GO TO 77
                  A=A*B/C**2
          $
01) LABEL
                  WRITE (3,77)
                           $
01) ILLEGAL LABEL
              77 FORMAT (17H DATA UNACCEPTABLE)
                         $                              $
01) ILLEGAL LABEL         02) SYNTAX
                  STOP
          /*
          $$
01) SYNTAX                02) NO END CARD
                                                UNDEFINED LABELS
00001       00003
```

All right, now let us see whether we can understand these messages. The listing of the program is indented on the page, 12 spaces in, as can be seen for example by looking at the position of the "C" in the very first line (the listing of the comment card). So anything which appears on the extreme left side of the output page is a diagnostic message, not part of the listing of the FORTRAN program.

When an erroneous statement is found, this particular compiler places a "$" symbol in the line below, immediately below the position at which the compiler noticed that something was wrong. (This is *not necessarily* the position of the actual error). The line after that contains a message describing the kind of error detected.

Now let us look at the actual messages, line by line.

The READ statement has no closing bracket after the second "1". The compiler was still searching for the closing bracket when it found the character "A" instead. Hence the marker symbol "$" appears underneath the "A", and the accompanying diagnostic message "SYNTAX" informs us that the FORTRAN statement runs afoul of the rules of FORTRAN grammar. "Syntax" means "the grammatical arrangement of words in a statement". In the English language, "The sky is blue" is correct, but

"Sky the is blue" violates the rules of English syntax. In FORTRAN, the statement READ (1,1 A,B,C violates the rules of FORTRAN syntax, because the identifier name "A" appears in a position where only a right parenthesis is permitted to appear. The character "A" is not wrong FORTRAN by itself; but it appears in a wrong position in the statement.

The FORMAT statement on the next line has given rise to two error markers "$", and correspondingly to two separate diagnostic messages. The diagnostic message prefaced by "01)" refers to the first "$" marker; the message prefaced by "02" refers to the second "$" marker.

The first message says "LABEL", meaning something is wrong with the statement label, or statement number (the two words are used interchangeably). In fact, there is no statement number, and that is the error: a FORMAT statement must have a statement number in order to be correct FORTRAN. The marker symbol "$" appears under the "T" of "FORMAT", because this is the point at which the compiler was first able to diagnose this particular error. Only after the full word "FORMAT" had been read, did the compiler know that this was meant to be a FORMAT statement; only then could it diagnose the absence of a statement number (label) as an error.

The second error message is "SYNTAX", with the marker pointing to the closing bracket of the FORMAT statement. The actual error is the absence of any decimal point in "4F10.0", but this error became evident to the compiler only after scanning the closing bracket: The compiler was still searching for the expected decimal point when it encountered the closing bracket instead. *It should be evident by now that the position of the marker symbol "$" must be interpreted with considerable care.*

The next error message is straightforward: The IF statement requires three statement numbers at the end, and only two statement numbers appear.

The next diagnostic refers to the statement after the GO TO statement. The message is "LABEL", indicating in this case the absence of a label (statement number) on a statement immediately following a GO TO statement. This statement could never be reached during program execution (see page 23).

The next diagnostic is well designed to mystify the programmer. The marker points to the statement number "77", and the message itself is "ILLEGAL LABEL". There is nothing illegal about 77 as a statement number, by itself! So what is wrong?

Well, statement number 77 in this position within a WRITE statement must be the statement number of a FORMAT statement (and in fact the FORMAT statement with statement number 77 follows immediately, and gives rise to the same diagnostic message concerning the statement number). But an earlier statement in the program, namely the statement "GO TO 77", has led the compiler to expect that statement number 77 will be an executable statement, a statement to which control can be transferred. FORMAT statements are not by themselves commands to the machine; rather, they inform the compiler how to translate certain other commands, namely the WRITE commands. It makes no sense to tell the machine "GO TO 77" when statement 77 is in fact a FORMAT statement. Once the statement number 77 has appeared in "GO TO 77", that statement number is an "illegal label" for a FORMAT statement.

Let us go on: The FORMAT statement itself has given rise to two diagnostics, the first of which we have already explained. The second one is due to the fact that the message is 18 characters long, but has been prefaced by "17H" instead of "18H". Thus the final letter "E" is no longer considered part of the message. By the time the

compiler has seen the closing bracket, it knows that something is wrong with that "E", hence the diagnostic.

The card with the "/*" in columns 1 and 2 is a normal control card for this particular system, indicating the end of the translation (compile) phase. The two error messages arise from the fact that the FORTRAN statement END has been omitted from the FORTRAN program. Without the statement END, the compiler thinks there is more FORTRAN program to be translated, and looks for another FORTRAN statement. Considered as a FORTRAN statement, the card with "/*" is quite ungrammatical FORTRAN; hence the first diagnostic, "SYNTAX". The second diagnostic tells us what the first diagnostic should have told us, namely the true nature of the error.

Certain types of error can be detected only after the entire program has been looked at by the compiler, and the corresponding error messages therefore appear after the entire listing of the program. The "UNDEFINED LABELS" message is of this type: as a result of the statement "IF(N) 3,3" the compiler was alerted to expect a FORTRAN statement with statement number (label) 3 somewhere else in the program. The compiler has now reached the very end of the program, and there is still no statement with statement number 3. Thus, we have a missing statement number, or in this peculiar jargon, an "undefined label". The actual value of the missing statement number appears on the next line, where we see 00001 and 00003. Our discussion has explained the 00003. Can you explain the 00001?

It is apparent that the diagnostic messages produced by this particular compiler are sometimes not so easy to understand. While compilers differ from each other quite considerably in the clarity of these messages (we have picked a rather "obscure" one for our example), no compiler yet made by man issues diagnostic messages which are always clear to every programmer, particularly to beginning programmers. Learning how to live with, and make the best of, somewhat obscure diagnostic messages is an important part of learning how to program. One thing is fairly safe (though not 100 percent safe): If the compiler issues a diagnostic message, something is wrong with the program! (*) When faced with obscure messages, avoid panicking. Fix those errors which you can detect, and resubmit the program for another run. Quite often, fixing one error will make several other obscure diagnostics disappear, which did not seem to have any relation to that particular error as far as you could tell before. For example, suppose that the statement following the "GO TO 77" should have had statement number 3, from the flow diagram of your program. When you supply that statement number 3, in response to the diagnostic message "LABEL" appearing there, another diagnostic message will disappear automatically: namely the message declaring statement number 3 as an "UNDEFINED LABEL".

So far, we have been talking about possible obscurity of the diagnostic messages. There are special student compilers which go to considerable length to make the diagnostic messages explicit and meaningful to an inexperienced programmer. The compiler not only diagnoses an error in FORTRAN syntax, but tells what sort of syntax error it is. As an example, consider the following erroneous statement, with two plus signs adjacent to each other:

```
3000 Y=A++B
```

With the WATFOR compiler, this produces the printed output (assuming this was the

(*) Like all other programs, compiler programs also have "bugs", and occasionally diagnose errors in a perfectly correct FORTRAN program. But do not count on this for your own program!

fifth non-comment statement in the program):

```
      5          3000 Y=A++B
***ERROR***         SX-D
```

The "5" at the beginning of the first line is the *running sequence number* of this statement (which has no effect on program execution); the "3000" is the statement number assigned to this statement by the programmer. The next line is the diagnostic message, giving the error code SX-D. These error codes must be looked up in a list (the list of error messages for your own compiler appears in Chapter XVI).

In WATFOR, the message SX-D stands for "Syntax Error - Missing Operand or Operator". The compiler expected an operand after the first plus sign, and instead found another plus sign. The statement Y=A+G+B, with the extra operand "G" after the first plus sign, is a grammatically correct statement. The actual statement, Y=A++B, has one operand missing.

This error message is beautifully explicit. However, there is a danger in messages of this kind. The true error might not be what the compiler thinks it is! For example, it is entirely possible that you *meant* to write Y=A+B, and your error consisted of putting an extra plus sign onto the card. In that case, it is not true that there is either an operator or an operand missing; in fact, there is one operator too many!

The machine can not, and does not, read your mind! The diagnostic message tells you that an error has occurred, and gives one possibility for the nature of that error. But it is up to you to look at your own program and determine what really went wrong. No one else can do it for you, and experience at doing this very thing is the main skill required of a successful computer programmer. Any fool can write a program; but it takes a man to get the bugs out of it!

Section C: Execution Time Error Messages.

After the FORTRAN program has been translated into machine language, a control card of some sort signals the end of the compile phase. Execution of the object (translated) program commences now. As long as everything goes all right, the orders obeyed by the machine are the translated versions of the orders in your FORTRAN program. But when something goes wrong (and lots of things can go wrong!), a supervisor program (sometimes called a monitor program or an operating system) takes control. An error message is printed out, and usually program execution is terminated altogether. The next student job is called in for translation and execution.

An example of the kind of error that can be detected only in execution of the object program is an incorrectly prepared data card. For example, the data card might contain the symbol "*" by mistake, in place of the minus sign "-". When the time comes (in the execution phase) to read this particular data card, the supervisor program is alerted to the fact that a character "*" appears in a field of the card which has been declared to contain a number. Such a field may contain decimal digits, a decimal point, and a "+" or "-" sign, but not the character "*". The error message produced by the supervisor program differs from machine to machine. Under Disk Operating System/360 this particular error message reads

IJT223I

where the symbols "IJT" at the beginning and "I" at the end indicate that this is a

message from the operating system (rather than from the FORTRAN compiler, for example), and the number "223" is the code number of the message itself. This code number must be looked up in a list (the list for your particular operating system appears in Chapter XVI, also). This particular message is listed as meaning:

> "An input or output record for which a FORMAT statement has been specified contains an illegal character."

Knowing the nature of the error, the message is clear enough.

However, usually we do *not* know the error to start with, and have to deduce the error from the error message. This particular message (as well as a lot of others from this operating system) may leave us a bit in the dark. First of all, we are not told whether the error occurred during input or during output. Both can happen: the offending operation might have been the result of an incorrect FORMAT statement for output to the line printer! Second, if we are using several sources of input (cards and magnetic tape, for example), we are not told which input medium produced the wrong input. Even if only a card reader is used for input, we are not told which particular READ statement in our program was being executed at this moment (there might well be several READ statements in the program). Finally, and this is unfortunately true of very many operating systems, we are not given the card image of the faulty card, on the output page.

Thus we shall have to do a bit of sleuthing to deduce that it was reading (rather than writing), that it was a faulty card (rather than a tape record, for example), which card it was, and which character on the card caused the trouble. Quite an order!

Here again, careful planning of the original program pays huge dividends. The practice of *immediate echo-checking of all input data* is recommended highly. As soon as some numbers are read in, they should be printed out on the line printer, before doing any computations at all. Then we can tell, merely by looking at the printed output, what card was read correctly last; the faulty card is of necessity the card right after that one.

Besides incorrect data cards, a frequent source of execution time error messages is an *illegal operation,* like attempting to divide by zero, or taking the square root of a negative number, etc. The compiler, looking only at the source program, is in no position to predict such happenings. They can be, and are, detected as errors only in execution time. Program execution may, or may not, be terminated as a result of such an error. For example, the error message for division by zero on the IBM Disk Operating System/360 is.

```
IJT225I     xxxxxxxFxxxxxxxx
```

where the characters indicated here by "x" are of little help to the programmer. But the eigth character of this set of 16 characters, the character "F" in our case, gives the nature of the error, to wit: "Floating-point divide exception". With this particular operating system, attempted division by zero does not terminate program execution; the computer goes right on. Unless you have output of debugging information throughout your program, it may be very hard for you to tell just where in the program the attempted division by zero occurred. Usually, the only thing to do is to put a lot of messages into the program, of type:

```
      WRITE (NPRNT,2101)
 2101 FORMAT (23H PASSED STATEMENT 2100    )
```

Such a message, placed right after statement 2100 of the program, results in output which tells you that you have passed this particular checkpoint, whenever you do pass it. (An alternative technique, using subprograms, is described in Chapter XIII). The divide-by-zero error message from the operating system will appear between two such checkpoint printouts, and in this way you can narrow down where in your program it happened.

Having narrowed down the fault to a certain area of your program, you take out the irrelevant checkpoint messages, insert more such messages into the suspect area, and resubmit the run. Usually, two runs with checkpoint messages suffice.

Here again, special student compilers and student operating systems are much kinder to the programmer. For example, the WATFOR error message for an attempted division by zero is: KO–2 ("Floating Point Division by Zero"), followed by a line saying: "PROGRAM WAS EXECUTING LINE ... IN ROUTINE M/PROG WHEN TERMINATION OCCURRED". Program execution is terminated immediately. The line number in this message refers to the running numbers which the WATFOR compiler has assigned to the FORTRAN statements of your program. So it is not necessary to do a lot of sleuthing to find out where the trouble occurred: the supervisor associated with WATFOR gives enough information to tell you that. But even WATFOR does not print out the card image of an offending data card, so it is still necessary to go back to the input deck to find which card it was, and which character on it caused the trouble. Thus, immediate echo checking of all input data numbers is still essential.

PROGRAMMING EXERCISES FOR CHAPTER V.

1. Prepare input cards for the program on page 47. Submit this job under your own system, and make sure that you understand every diagnostic message on the printout (you will require the material in Chapters XV and XVI for this).

2. The following FORTRAN program will cause an attempted division by zero in execution, but will not cause a compile-time diagnostic:

```
1 FORMAT (7E16.6)
  NPRNT=3
  A=1.0
  B=0.0
  C=A/B
  WRITE (NPRNT,1) A,B,C
  STOP
  END
```

   Prepare this program as a job under your system, and run it; inspect the execution time error message and learn to recognize it.

3. Write programs similar to that of exercise 2, and submit them, with the following known faults in them:

   a) Raise the number zero to an exponent which is zero
   b) Raise the number zero to an exponent which is negative
   c) Specify the card reader's number in a WRITE statement
   d) Specify the line printer's number in a READ statement
   e) Supply a data card with a known incorrect character on it
   f) Attempt to read data cards when there are no data cards supplied
   g) Deliberately omit the control card which signals the start of the execution phase.

   Prepare all these as computer runs, and inspect the printed output. *(Note:* This exercise will save you a *lot* of trouble and sweat later on!)

PROGRAMMING EXERCISES FOR PART A

Note: We recommend highly that you complete most of these exercises before commencing study of Part B of the book, which starts with Chapter VI.

1. The volume of a cylinder is $V=pr^2h$ where r is the radius, h is the height, and p=3.14159. Read R and H from a data card, compute V, and print out R, H, and V. Keep reading data card after data card, until a blank data card is encountered; then stop.

2. Print a table of values, from 1.0 to 25.0, and their squares. *Hints:* No data need be read in; the program is written as a loop. The loop variable is set initially to VALUE=1.0, and the first statement within the loop is SQUAR=VALUE**2; we then print out VALUE and SQUAR; we increment VALUE by the statement VALUE=VALUE+1.0, and we test whether we should go around the loop once more; it is best to test VALUE against a number such as 25.5, rather than against 25.0, because of the possibility of error accumulation (explain!).

3. Same as 2, but the table should contain not only values and squares, but also cubes, fourth powers, and fifth powers, in a five-column table. *Hint:* Compute and print out one line of the table at a time. Use statements such as SQUAR=VALUE**2, CUBE=SQUAR*VALUE, and so on. Do not use a statement such as VALUE=VALUE**2, since this has the effect of overwriting the content of the storage location VALUE, thereby upsetting the operation of the loop.

4. A man takes a job for 35 days. His pay for the first day is $0.01. His pay for the second day is double that, or $0.02. Each new day, his pay doubles, so that he gets $0.04 on the third day, $0.08 on the fourth day, and so on. The program should print out a table (if you are ambitious, supply a table heading also) with three columns: the number of the day in column 1, the day's pay in column 2, and the total pay so far in column 3.

5. Write a program to read four numbers W, X, Y, Z from a data card, find the biggest of the four, store that in location BIG, and print out all five numbers. You will have to make repeated use of the IF statement in this program.

6. Read the coefficients A, B, C, D, E, F in the pair of linear equations

   A*X+B*Y=E
   C*X+D*Y=F .

   then echo-check. Determine the values of X and Y, and print them out. *Note:* Guard against the degenerate case, so-called, where DET=A*D-B*C vanishes; otherwise you generate a division by zero. In this program, stop execution in this case.

7. Write and debug a more elaborate payroll program. The input data card for each employee has four numbers on it, namely his hourly wage WAGE, the number of straight hours he worked (HOURS), the number of overtime hours OVTIM, and the percentage of his gross wage which he wants deducted as a savings deduction (SVPCT). Besides this savings deduction, there is a taxation deduction and a superannuation deduction, both depending upon the hourly wage rate, according to the following table:

Hourly Wage	% tax deduction	% superannuation deduction
1.50-1.89	5.0	2.0
1.90-2.49	7.5	2.5
2.50-3.74	9.7	3.5
3.75-5.00	15.5	5.0

   An hourly wage less than $1.50 or greater than $5.00 indicates an error on the data card. Other things to look out for are: (i) HOURS must not be negative or more than 40.0; (ii) OVTIM must be zero unless HOURS equals 40.0, and OVTIM must not exceed 12.0; (iii) the savings deduction SVPCT must not be negative, and must not exceed 20.0 percent. All input cards must be tested for all these things, and appropriate error messages must appear in the payroll table whenever one or more of these conditions is violated.

   The payroll table itself should contain the input numbers for each employee, then columns for gross pay, total of all three deductions, and net pay, respectively (no more than seven numbers can go on one line with our present standard FORMAT statement). Stop on a blank card.

PART B

In this, second, part of the book, we present the USA Standard Basic FORTRAN IV language in full. There are many dialects of FORTRAN IV, but all of them include USA Standard Basic FORTRAN IV. That is, a program written entirely in USA Standard Basic FORTRAN IV will compile and work with every FORTRAN compiler.

Furthermore, while more elaborate versions of FORTRAN certainly exist, and contain nice features, the facilities provided by USA Standard Basic FORTRAN IV are sufficient for the vast majority of applications.

Chapter XVII, in Part C of this book, contains a list of the additional facilities provided by your particular version of FORTRAN.

CHAPTER VI

INTEGERS AND REAL NUMBERS IN A COMPUTER

Section A: Introduction and FORTRAN conventions.

In Chapter II, page 17, we gave a brief introduction to the concept of integers (counting numbers) and real numbers (measuring numbers); but so far we have worked mainly with real numbers.

There are many applications for which integers are the natural tool. In fact, this is so whenever we wish to count things, or to number items in a list of things. If a part in the stock-list is part number 241, we do not mean part number 241.003. If we wish to go around a loop in our program 45 times, this number is a counting number, that is, 45 exactly, not perhaps 44.999 or 45.001. It makes sense to ask for the 56th name in a list of names, but it makes no sense to ask for the 56.73th name in the list.

A computer stores integers in a different way from real numbers, and carries out different operations on integers and real numbers. In that respect, computers are very different from us. We write down integers (like 35) in pretty much the same way as real numbers (like 35.14), and when we add two numbers, say, we do not much care whether these two numbers are both integers, or both real numbers, or mixed. But a computer has two quite different "arithmetic units" (units which perform arithmetic operations like addition, subtraction, multiplication, and division), one for integer arithmetic, the other for real number arithmetic.

FORTRAN uses the same symbol "+" for addition of integers and for addition of real numbers. The FORTRAN compiler inspects the arithmetic expression as a whole to see which kind of addition is wanted, and translates the symbol "+" into the machine command for integer addition in one case, and into the machine command for real number addition in the other case. *Addition of an integer to a real number is simply forbidden:* the arithmetic expression is declared erroneous, being of "mixed mode".

In a FORTRAN program, numbers appear in two forms: (i) explicitly, as information to be converted directly into machine form, and (ii) implicitly, as the contents of certain named memory locations. For example, the arithmetic assignment statement

```
WIDTH=WIDTH+1.5
```

contains the explicit number 1.5, which must be added to the content of the named memory location WIDTH, and the result must be stored back into that location.

Explicit numbers are called FORTRAN *constants,* whereas named memory locations are called FORTRAN *variables.* In order to enable the FORTRAN compiler to tell what is meant by the programmer, there are certain standard conventions to distinguish integer constants from real number constants, and to distinguish integer variable names from real number variable names.

A FORTRAN *integer constant* is an ordinary integer, *without a decimal point,* preceded by either a blank, a plus sign "+", or a minus sign "-". If there is a blank in front, the integer is said to be *unsigned* and its value is taken to be positive.

Examples: The constants 31, -452, +76982, +0, -0, 89 are all integer constants. The first and the last are unsigned.

When such a string of characters is encountered by the FORTRAN compiler, it is converted automatically into proper machine form for an integer, and stored away inside the machine. However, there are limits to the maximum size of integers which any particular computer can handle. The biggest integer permitted in an IBM/360 computer is the number 2147483647; integers larger than this can not be stored into machine memory, and are said to give rise to "integer overflow". The maximum size of integers depends very much on the particular machine. For example, it is equal to 140737488355327 in the CDC 3600. The maximum size of integers for your version of FORTRAN is given in Chapter XVII.

A FORTRAN *real constant* appears in one of two forms:

i) an ordinary decimal number *with a decimal point,* preceded either by a blank, a plus sign "+", or a minus sign "-";

ii) an ordinary decimal number, *with a decimal point, followed by the code letter "E",* followed in turn by the number of places the decimal point is meant to be shifted (positive for a shift to the right, negative for a shift to the left, zero for no shift at all); the whole is preceded by either a blank, or a plus sign "+", or a minus sign "-".

We have encountered both these number forms already, the form (i) for input numbers on data cards, the form (ii) for output numbers from our WRITE statements.

Examples: The real number 436.72 can be written in FORTRAN as 436.72, 4.3672E+02, 4.3672E+2, 4.3672E2, 0.43672E+03, 0.43672E+3, 0.43672E3, 43672.0E-02, 43672.E-2, 436720.0E-3, etc. etc. The real number "one" can be written as 1.0, 1., 1.E0, 0.1E1, etc. etc., but *not* as just 1 by itself (this would be the *integer* "one", not the *real number* "one"; the two do not look the same inside the machine).

The following are examples of invalid FORTRAN constants:

3,471 is invalid because it contains an imbedded comma
3,471.3 is invalid because it contains an imbedded comma
2.E is invalid because nothing follows the letter "E"; write 2.E0 or 2.0 instead.

Just as there are limitations on the maximum size of an integer in any given machine, so there are limitations on the number of places the decimal point of a real number can be shifted, i.e., on the maximum number of right shifts or left shifts following the code letter "E". For example, no more than 75 is allowed in the IBM/360, whereas the number of shifts can be as high as 308 in the CDC 3600. The limitation for your particular computer is stated in Chapter XVII.

All right, so much for the way in which the FORTRAN compiler distinguishes between integer constants and real number constants.

When it comes to variable identifier names, i.e., the names of memory locations in the machine, the rule used by FORTRAN is exceedingly simple. The compiler looks at the *first letter of the identifier name:*

Identifier names starting with any of the letters I, J, K, L, M, or N are the names of *integer variables* (memory locations where integers are stored). Identifier names starting with other letters of the alphabet are the names of *real variables* (memory locations where real numbers are stored).

Thus MONEY, KLOT, JOKER, and LOAF are memory locations for storing integers. CENTS, DOLLR, WIDTH, DOPE, and WOW are memory locations for storing real numbers. If the usual word for a certain quantity has the wrong starting letter for this rule, the programmer simply supplies a suitable letter in front. Thus, the English word "read" has the wrong starting letter for an integer variable. Hence we used the variable name NREAD for the (integer) number assigned to the card reader. Similarly, if we wish to store a certain number of cents as an integer, we may decide to use the name KENTS instead of CENTS, or perhaps NCNTS - use your own imagination!

The use of names with particular starting letters is hard to get used to at first, and beginning FORTRAN programmers make quite a few mistakes because of forgetting this rule. But after a while, this convention becomes entirely second nature to the FORTRAN programmer.

Section B: Implied Conversion.

It is frequently necessary to convert integers to real number form, or vice versa. The simplest way to do so is by use of the ordinary arithmetic assignment statement. We recall (see page 18) that the assignment statement has the form

NAME=wfae

where *NAME* is the identifier name of some memory location, and *wfae* is any well-formed arithmetic expression.

The arithmetic expression to the right of the equal sign must be all of one type, i.e., all things in it must be integers *or* all things in it must be real numbers. This arithmetic expression is evaluated first of all, before any storing is done.

However, the starting letter of the identifier name to the left of the equal sign is allowed to indicate a quantity of opposite type. For example, the assignment statement

A=K+5

is grammatically correct FORTRAN. If the content of memory location K at this moment is 7, say, then we add 5 to it to get the integer 12. Then we look where this is to be stored. We see that the name of that memory location, i.e., the name A, is a real number identifier name. Therefore we convert the integer 12 to the real number 12.000, and we store that real number into memory location A. This process is called *implied conversion* from integer form to real number form.

Implied conversion in the other direction, from real number form to integer form, is also permitted, but must be handled with care since it may alter the value of the quantity. For example, consider the statement

JOKER=2.998

The arithmetic expression on the right side of the equal sign is just a single real number (this is quite permissible, see the rules in Chapter II); but when we look at the name of the memory location into which this number is to be stored, we see the name JOKER, which is an integer identifier name. Thus only an integer can be stored there.

Any reasonable person would "round up" 2.998 to the nearest integer, 3; but this is *not* what FORTRAN does. Rather, FORTRAN *ignores everything after the decimal point* (this is called *truncation* of the number), and thus the value stored into location JOKER is 2, not 3! Similarly, -2.998 is truncated to -2, *not* rounded to -3.

As long as you allow for this peculiarity of FORTRAN, implied conversion from real number form to integer form is quite all right. But do not forget it!

Section C: Integer Arithmetic.

Integer addition, subtraction, and multiplication are exact. The only thing which can go wrong is that the result might be too large to fit into this particular computer. An integer overflow execution time error message is issued by the operating system when this happens. Either program execution is stopped altogether, or the resulting numbers are likely to be meaningless from this point on.

Division of one integer by another is not always possible, however. For example, 27 can not be divided by 4; 4 goes 6 times into 27, but leaves a *remainder* of 3.

In ordinary calculations, we do not worry about this. We write 27/4=6.75 and leave it at that.

FORTRAN, however, insists on getting an integer answer (even if the answer is not an integer), and it does so by the simple process of *ignoring the remainder.* That is, the FORTRAN answer for the division of 27 by 4 is the integer 6; for example, the FORTRAN statement

```
JILL=2+27/4
```

has the effect of adding 2 and 6 to get 8, which is stored into memory location JILL. Similarly, 3/4 is the same as 0 in FORTRAN! And -3/4 is equal to 0, also!

Peculiar as this FORTRAN rule is, it can be used to good advantage. For example, suppose we have two numbers I and J in the machine, and we want to test whether I is divisible by J, without remainder. To do this, we construct the quantity K by the command

```
K=I-(I/J)*J
```

We assert that K is zero if and only if I is divisible by J without remainder. To see this, let us look at two special cases, one where I is exactly divisible by J, the other where I leaves a remainder after division by J. As our first test case, we take I=24 and J=4. In this case, both the ordinary value and the FORTRAN value of (I/J) is 6, and thus K=I-6*J=24-6*4=0; thus K is indeed zero in the machine. As our second test case,we take I=27 and J=4 (so that I is not divisible by J). The ordinary value of 27/4 would be 6.75, but the FORTRAN value of (I/J)=(27/4)=6, and thus K=I-6*J=27-6*4=3, so that K is indeed different from zero in the machine. Quite generally, K is zero if and only if I is exactly divisible by J, without remainder.

Section D: Real Number Arithmetic.

With real numbers, division is no problem (except for division by zero), and arithmetic overflow (numbers too big for the machine to handle) is rare, and usually due to

some bug in the program. But there are other problems, arising from the fact that real numbers in the machine are only approximate, not exact; i.e., each machine keeps only a certain number of decimal places, that number differing from machine to machine.

In Chapter III, we developed a square root program, and the output from this program was given on page 31. Column 3 of the output table was the ratio (P-Q)/Q, where P is the input approximation to the square root, and Q is the improved approximation. This ratio becomes progressively smaller as the accuracy of the input number P improves. When the ratio becomes less than some preset tolerance level, the approximation is considered good enough.

We now reproduce column 3 from page 31, containing the successive values of this ratio, from two computer runs: (1) On a CDC 6400 computer, and (2) On an IBM/360 computer. The FORTRAN programs are identical.

VALUES OF THE RATIO (P-Q)/Q

Iteration	*CDC 6400*	*IBM/360*
1	-0.322581E-01	-0.322579E-01
2	0.520562E-03	0.520485E-03
3	0.135493E-06	0.307797E-06

The results of the first iteration agree reasonably well, the difference being 2 in the last place, or 1.5 parts in 100000, approximately. But the numbers from iteration 3 are completely different.

The IBM/360 real numbers are stored to an accuracy of approximately 7 decimal places, whereas the CDC 6400 real numbers are stored to an accuracy of approximately 14 decimal places. In the third iteration, the values of P and Q are:

```
   P=3.09838709677
   Q=3.09838667697
  ----------------
P-Q=0.00000041980
```

When we divide this difference by Q, we get the CDC 6400 number. But in the IBM/360, the values of P and Q are kept only to seven significant figures; thus the subtraction looks like this:

```
   P=3.098387
   Q=3.098386
  -----------
P-Q=0.000001
```

Even though both P and Q are accurate enough for most practical purposes, the process of subtraction has resulted in a value which is more than twice the correct result. We have lost all significant figures by this subtraction. When this incorrect difference is divided by the value of Q, we obtain an incorrect output number such as the one shown in the table. (*)

Errors arising in this way are called *round-off errors,* and they are *unavoidable* in any calculation with real numbers (as opposed to calculations with integers). In

(*) To get the precise number in the table, it would be necessary to carry out these operations in binary arithmetic, not in decimal arithmetic. We have not done so because this would merely complicate the discussion, without adding anything of value.

the example shown here, 14-figure precision is sufficient to make these round-off errors unimportant, whereas 7-figure precision results in complete loss of significance. But round-off errors also occur with 14-figure precision, and there are more elaborate calculations for which even 14-figure precision is inadequate.

In purely commercial data processing, where most of the work is input and output rather than mathematical calculation, 7-figure precision is sometimes sufficient. But in most scientific and engineering calculations, 7-figure precision is highly suspect.

In Basic FORTRAN IV, there is only one kind of real number storage, with whatever precision is provided for such numbers in the machine in question. But the full FORTRAN IV language includes provision for "double-precision real numbers". It is possible to specify that certain constants, and certain variable identifier names, are to correspond to double-precision numbers in the machine, and to carry out arithmetic operations to this double precision. If basic precision on the machine is only 7 figures, practically all scientific and engineering calculations on that machine require double precision working. But if the basic precision is 14 significant figures, then double precision work is required only rarely.

When deciding between machines, it is important to have a clear idea what the machine is supposed to do. Generally, the requirements for commercial data processing and for scientific (or engineering) calculations are quite different, and a machine which is good for one is a dubious choice for the other. For commercial data processing, the quality and reliability of the input-output equipment is of major importance, and so is the amount of auxiliary storage available for extended commercial files (tapes, disks, data cells, etc.). Neither speed of arithmetic operations nor precision of the real numbers stored in the machine matter very much. Exactly the opposite holds for scientific and engineering calculations.

The USA Standards for FORTRAN and for Basic FORTRAN say nothing about the actual (or even a minimum) precision to be associated with "single precision" and "double precision" real numbers. Nor do they say anything about speed, reliability, etc., of input-output equipment, size of file storage available, etc. Yet these considerations can not be ignored in actual computing practice.

So far, we have discussed possible loss of precision in a single arithmetic operation (subtraction). There is a quite separate question, concerning the way small errors made in any one arithmetic operation add up to big errors after a lot of arithmetic operations have been performed. This is called *error propagation,* or, by more pessimistic programmers, *error multiplication.*

To understand error propagation, it is essential to understand the difference between *rounding* and *truncation* in operations with numbers of limited precision. As an example, consider addition of the two numbers 12.36483 and 0.04351878:

```
 12.36483
+0.04351878
-----------
 12.40834878
```

If we are to store this number to an accuracy of seven significant places, the last three digits can not be stored. We now have two choices:

1. *Truncation:* Ignore these figures, hence store 12.40834
2. *Rounding:* Go to the nearest seven-figure number, i.e., store 12.40835

It is clear that the answer which is eventually stored away after this one addition is not very different.

But there is an enormous difference in the way these little errors add up to big errors. With truncation, the errors all tend to go in the same direction, and hence add to each other. If we make an error of one part in ten million *each time,* then after ten million additions we have no accuracy left at all. On present computers, ten million additions is only about *one minute* of computing time (less than this on the really fast machines)!

On the other hand, with rounding the errors do not all go in the same direction. When we "round up" (as in the example above), the number stored is bigger than the true answer; when we "round down", the number stored is smaller than the true answer. Since we are as likely to have to round up as to round down, in any one calculation, the individual little errors tend to cancel each other.

The difference between truncation and rounding is like the difference between a sober man and a drunk: the sober man takes one step after another, always in the same direction; the drunk is as likely to step backwards as forwards. After a million steps, the sober man is one million steps away from his starting point. But the drunk is as likely to be to the left of the starting point as to the right, and his distance from the starting point is not likely to be anywhere as large as a million steps. In fact, it is likely to be somewhere around one thousand steps only (one thousand is the square root of one million). The only thing peculiar about this analogy is that, when it comes to error propagation, we greatly prefer the drunk. We do not want the answer to be many steps away from the true answer!

The FORTRAN language specifications do not state whether real number arithmetic should be done with truncation or with rounding. This is an unfortunate omission. Rounding is very much preferable to truncation. Indeed, the presence or absence of rounding can make the difference between a reasonable answer and a string of meaningless digits out of the machine.

Besides the possibility of overflow, i.e., real numbers too big for the machine to handle, there is the opposite possibility of *underflow.* For example, in IBM/360 FORTRAN the magnitudes of real numbers can not be larger than about 10^{75} (a digit "1" with 75 zeros after it); they can not be smaller than 10^{-75} (a decimal point, then 74 zeros, then a digit "1"). All numbers smaller than 10^{-75} can only be represented by the number zero in the machine. Replacement of a small number by zero in the machine is called "underflow".

Arithmetic underflow does occur at times, but usually it is not really a mistake. When the numbers get to be so very small, not only must they be replaced by zero, but replacement by zero is exactly the right thing to do in such a case.

Nonetheless, some systems programmers (people who write these "operating systems" or "monitor" programs) have seen fit to produce supervisor programs which declare such an underflow to be an error. Program execution is interrupted, or perhaps even terminated, and an execution time error message is issued.

In this author's view, this is an entirely undesirable practice. Arithmetic underflow is not an error in itself. It may, or may not, be a symptom of a real error somewhere. It would be perfectly sufficient to provide an *option* by which a programmer can choose to be warned of underflows.

Section E: Raising a Number to Some Power.

Besides the elementary operations "+", "-", "*", and "/", the FORTRAN language

also provides the operator "**" for *exponentiation* (i.e., raising a number to some power). For example, J**3 is the same as J*J*J, B**4 is the same as B*B*B*B, D**0.5 is the same as the square root of D, and so on.

If the exponent (the number *after* the symbol "**") is an *integer,* then the FORTRAN compiler translates the expression into an instruction to the machine to carry out successive multiplications. This is true no matter whether the quantity *before* the "**" sign is integer or real.

But if the exponent is a *real number,* then the quantity *before* the "**" sign must also be a real number (not an integer), and *it must not be negative.* The reason for this rule is that the compiler translates A**3.0 very differently from A**3 without a decimal point on the number 3. A**3.0 is translated into a set of instructions for (i) taking the *logarithm* of A, (ii) multiplying this logarithm by the real number 3.0, and (iii) finding the anti-logarithm of the result. Don't worry if you have forgotten all about logarithms; this does not matter. What does matter is that this set of instructions is much more complicated, takes considerably longer to carry out, gives lower accuracy, and fails altogether if A is negative (for then the logarithm of A is not a real number).

Thus, *if the exponent you want is actually an integer, be sure to write it that way.*

Certain special cases lead to trouble with exponentiation: (1) zero to the power zero is undefined, and (2) zero to a negative power is undefined. Such troubles show up only in the execution phase, when the time comes to perform these operations. Bugs of this sort are not caught by the compiler. Generally, if the exponent is not a positive integer constant (but rather an integer variable name, or a real number), it is desirable to write IF statements into your program before the exponentiation statement itself. If the exponent is negative and the number is zero, or if the exponent is real but the number is negative, then the exponentiation statement should be bypassed, since it would cause trouble in execution.

Summary and Formal Definitions for Chapter VI.

1. *Constants:*

   Integer: +, -, or blank, followed by digits; no decimal point. Signed (+ or -) and unsigned integer constants.

   Real numbers:

   Basic real constant: +, -, or blank, followed by digits with a decimal point included.
   E-type real constant: Basic real constant, followed by letter "E", followed by an integer constant representing the number of shifts of the decimal point (positive for shift to the right, negative for shift to the left).

2. *Variable Identifier Names:*

   Integer variables: First letter is one of I, J, K, L, M, or N.
   Real Number Variables: First letter is any letter other than one of the above six letters.

3. *Implied Conversion:*

   If the variable name to the left of the equal sign in an arithmetic assignment statement has a type different from the type of the arithmetic expression to the right of the equal sign, then a conversion operation occurs before storing the value. Conversion to an integer involves truncation.

4. *Precision, Round-off Error, Rounding:*

   The precision of a real number is the number of significant decimal digits. Round-off errors occur because of limited precision. Error accumulation is worse for "truncation" than for "rounding".

DRILL EXERCISES FOR CHAPTER VI.

1. Explain what is meant by mixed mode, and what Basic FORTRAN IV does with mixed mode expressions.

2. What is the meaning of a "FORTRAN constant" and of a "FORTRAN variable"?

3. State which of the following are valid integer constants, and correct the invalid ones.

a)	0	b)	79	c)	0.
d)	4,572	e)	-265	f)	-265.000

4. State which of the following are valid real number constants, and correct the invalid ones.

a)	0	b)	0.	c)	-0.
d)	-88.9999	e)	4,572	f)	4,572.0
g)	4572	h)	3.E	i)	45.9E-2
j)	3E0	k)	3E	l)	4,572.0E-02

5. Write in usual form the values represented by the constants below:

a)	5.27E0	b)	5.27E+3	c)	5.27E-3
d)	.527E-2	e)	.00527E+6	f)	99999.E-5

6. The following FORTRAN real number constants are intended to express the ordinary numbers given next to them. State which are correct, and correct the errors in the others:

a)	578E-5	.00578	b)	.578E+1	57.8	c)	.243E+3	-243
d)	243	243	e)	75.9E	75.9	f)	4.2E-5	.000042

7. Which of the following are incorrect integer variable identifier names, and why?
NUMBER, JACK, BOY, LOTTERY, *MAN, MYGIRL, MAMA, PAPA, MY MAN, MYNAH

8. Which of the following statements involve implied conversions? Which of these conversions, if any, give results different from the usual rounding of numbers?

a)	WIDTH=3.87	b)	WIDTH=38	c)	KLOT=3.87
d)	KLOT=38	e)	KLOT=4.12		

9. Assuming the numerical values I=2, J=2, K=7, and L=-4, evaluate the following arithmetic expressions:

a)	I*(J-K)/(8+L)	b)	(I*(J-K)/8+L)
c)	I*((J-K)/(8+L))	d)	(I*(J-K))/8+L

10. With I, J, K, L as in problem 9, and A=2.0, B=3.0, state for each expression below whether it is grammatically correct FORTRAN; if not, why not; if so, state its value.

a)	I*A/J+L	b)	I*K/J+L	c)	I*K/J+B
d)	A**I+B**K	e)	B**A+A**B	f)	I**J+J**K
g)	I**J**K	h)	A*B**J	i)	A**B*J
j)	A**B**J	k)	A**B+B**J	l)	A**B+L**J

11. Detect the errors in the following program segment:

```
      A=-1.0
      B=0.0
      C=1.0
      I=1
1100  Y=(I-5)**A
1200  YNOT=A**(I+5)
1300  YOYO=B**(I-4)
1400  YOY=I-3
1500  SOD=A**YOY
1600  TOY=B**YOY
1700  ZUY=C**YOY
```

CHAPTER VII

MORE ABOUT INPUT AND OUTPUT: THE FORMAT STATEMENT

Section A: Introduction.

The "cookbook" approach of Chapter IV works only as long as all numbers to be read and printed are real numbers. When we wish to read or print out integers, the (unexplained) "standard" FORMAT statements of Chapter IV fail. In this Chapter, we shall take another, more careful and systematic, look at the FORTRAN commands for reading, and printing out, information.

There are many kinds of input devices and output devices attached to computers. For example, the card reader is purely an input device, the line printer is purely an output device. Commonly, another pure output device is attached to the computer also, namely a computer-controlled card punch; blank cards are fed in, and the computer controls the way holes are punched into these cards. Besides these devices, there are usually some magnetic tapes, or magnetic disks, or both, which can be used as both input devices (information can be read from magnetic tape into the main memory of the computer) and output devices (information from main memory can be written onto a magnetic tape). Programming for these other devices will be discussed in Chapter XI; here we discuss programming for the line printer and the card reader.

In order to organize printing of a line on the line printer, we must tell the machine a number of things:

(a) from which memory locations in main memory are the numbers to be taken?
(b) that the printer (rather than, say, a magnetic tape) is to receive the information;
(c) just what "layout" of the printed line do we want?

The FORTRAN statement for output has the form:

WRITE (*n*,*m*) *List*

where *n* is the number assigned to the output device to be used (for example, the number 3 may be assigned to the line printer in a given installation), *m* is the statement number of a FORMAT statement, and *List* is a simple list (see page 36) of memory location identifier names, stating the memory locations from which the output numbers are to be fetched. This list is the answer to (a) above; the number *n* is the answer to (b) above; but the answer to (c) is to be found, not inside the WRITE statement itself, but rather in the FORMAT statement with statement number *m*. This FORMAT statement determines *how* the numbers are to appear on the printed line.

Section B: Characters, Fields, and Field Descriptors.

A printed line consists of a number of characters, which may be alphabetic characters,

numeric characters, or special characters. The total number of characters which can be put on a printed line depends on the line printer used in your installation. Most line printers allow at least 120 characters on each line; some installations use line printers with up to 132 characters to a line, and so on. But if you want to write programs which can be run successfully in different installations, it is best to restrict yourself to lines with no more than 120 characters.

A single character is rarely enough to print an output number. For example, the number -23.54 requires the six characters "-", "2", "3", ".", "5", and "4". A group of adjacent characters on a printed line is called a "field"; within a FORMAT statement, we specify one field for each number which we want to print out, and (if we desire) some additional fields for messages.

The number of characters in a field is called the "field width". This field width must be large enough to allow printout of the number. For example, if we wish to print out the integer -5247 we require the five characters "-", "5", "2", "4", and "7", so our field width must be at least five.

To specify a field within a FORMAT statement we use a "field descriptor". For example, the field descriptor for an integer number in a field of width 15 is I15. The code letter "I" specifies that the number is an integer (rather than a real number) and the "15" is the field width. If the actual value of the integer is -5247, the number appears like this:

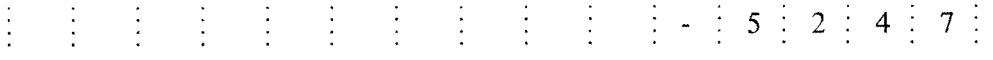


Figure 1: -5247 in an I15 field

The fact that the number itself is at the extreme right side of the field is called "right-adjusted". The remaining ten positions in the field are left blank (note that the blank is itself a "character").

If the field descriptor had been I12 (instead of I15), the field would have appeared as follows:

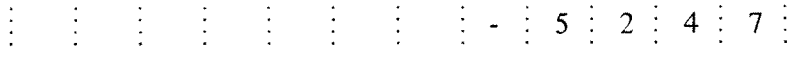


Figure 2: -5247 in an I12 field

Next, consider printout of real numbers. To print out a number of dollars and cents, for example 264 dollars and 75 cents, we want an ordinary decimal number, with two digits after the decimal point: 264.75. If we choose a field of width 10 for this number, then the FORTRAN field descriptor is: F10.2; here the code letter "F" specifies real number in ordinary decimal form, the number "10"is the field width, and the ".2" afterwards means that we want exactly two decimal digits to appear after the decimal point. The appearance of the field is:

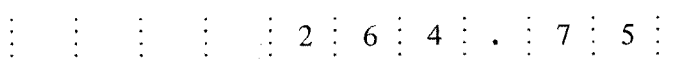


Figure 3: 264.75 in F10.2 field

This is the preferred form of printout for currency, but not necessarily for other real numbers. Suppose that the number 264.75 is a length of 264 and 3/4 inches, but that we want our lengths printed out to the nearest thousandth of an inch. We would then specify three, rather than two, decimal figures after the decimal point. If the total field width is the same, the field descriptor is now F10.3, and the printed field

looks like this:

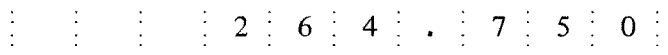


Figure 4: 264.75 in F10.3 field.

This form of printout is easy to read and familiar, but it becomes awkward for very large and for very small numbers (see page 8). If we have to print out numbers of all sizes, the preferred form is a mantissa followed by the number of places the decimal point is to be shifted, the so-called E-format. For example, let us specify the field descriptor "E16.6" for the same number. Here "E" is the code letter for this form of output, "16" is the field width, and ".6" means that the mantissa has six decimal digits after the decimal point (and a digit "0" before the decimal point). The result is:

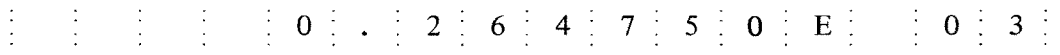


Figure 5: 264.75 in E16.6 field.

This is in fact the form of printout which we have used in part A of the book. Our "standard" FORMAT statement for printed numbers has been:

```
1 FORMAT (7E16.6)
```

The "7" in this specification is a *duplication factor*, saying that we are specifying seven identical fields, each with field descriptor "E16.6". Thus an equivalent FORMAT statement would be:

```
1 FORMAT (E16.6,E16.6,E16.6,E16.6,E16.6,E16.6,E16.6)
```

It is obvious that the use of a duplication factor makes life a lot easier.

At this stage, it might be interesting to discuss just how we came to choose this particular "standard" FORMAT. First of all, since we wanted to print out real numbers, we had to specify either the "F" or the "E" type of field. Not knowing the size of the numbers which might emerge, we had to use the E-format. In order to get six significant digits for the output numbers, we specified six digits after the decimal point of the mantissa (note that the "0" before the decimal point gives no real information); hence our choice ".6". The minimum field width for the output number is determined as follows: we need one position for the sign (if the number is negative; a space is left if the number is positive), eight positions for the mantissa, one position for the code letter "E", and three positions for the number of shifts. Altogether, this means thirteen characters. By specifying a field width of sixteen, we ensure some blanks in front (four blanks in front of a positive number, three blanks in front of the minus sign of a negative number); the blanks space the numbers apart from each other, for easy reading.

Note that our choice of six decimal digits after the decimal point of the mantissa, has given rise to a minimum field width of thirteen. More generally, *if we specify d decimal digits after the decimal point of the mantissa, then the minimum field width is d+7.* With this minimum field width, the numbers would not be spaced apart at all, so that a somewhat larger field width is highly desirable.

We must still explain why we chose a duplication factor of seven. Seven fields, each of width sixteen, means 7*16=112 characters altogether. This can be printed on one 120-character line (the extra characters at the end are simply left blank). But eight such fields would mean 128 characters, more than some printers permit on one line. 120-character lines are the minimum; 132-character lines are fairly common.

Besides printing out *numbers*, we have shown already in Chapter IV how to print out *messages*, and how to leave *blank fields*. The field descriptor "5X" specifies that five positions on the printed line should be left blank (i.e., a blank field of width five). The field descriptor "5HHOURS" specifies a field of width five which is to be filled with the five characters: "H", "O", "U", "R", and "S". For blank fields and character string fields like these, no number is fetched from machine memory. Rather, the contents of the field are specified completely by the field descriptor itself.

Section C: The Output Record and the Printed Line.

So far, we have looked at individual fields. Now let us look at the output as a whole. Compared to Chapter IV, the main changes are: (1) we can now print out integers as well as real numbers; (2) we can print out real numbers in forms other than E16.6; and (3) we can now interleave messages and numbers on a single line of output (whereas in Chapter IV we printed some lines with numbers only, and other lines with nothing but messages).

Furthermore, the FORTRAN statement "WRITE" is not restricted to output of printed information on the line printer. For example, we can direct the machine to produce *punched output cards* on the card punch attached to the computer. All we need to do is to specify, as the first number within the parentheses of the WRITE statement, the number assigned to the card punch, rather than the number assigned to the line printer. Since the card punch is a particularly slow-acting device, most installations discourage students from asking for punched output cards, and rightly so. But in data processing work, punched output cards are frequently the preferred form of output from the computer run, since such cards can be used later on as input data cards for the next run; whereas printed output can not (with present computer technology) be used for input to the computer.

No matter which output device is specified in the WRITE statement, the computer starts by looking at both the list within the WRITE statement, and the field descriptors within the FORMAT statement, and combines that information (in a way which we shall describe shortly) to produce a string of characters, kept in coded form inside the main memory of the machine. After the entire information for one printed line, or for one punched output card, has been assembled inside the machine, a command is issued to actually print that line, or actually punch that card. In either case, we say that we have produced one "output record".

Let us illustrate these points by a particular example, namely the following short FORTRAN program:

```
      NPRNT=3
      MONTH=8
      NGIN=526
      BLNCE=-9.07742
      WRITE (NPRNT,2300) MONTH,NGIN,BLNCE
 2300 FORMAT (17H PROJECT MOONBEAM/10H REPORT NO ,I3//I5,
     110H ENGINEERS/15H BANK BALANCE =,F10.3,11H  MEGABUCKS)
```

The FORMAT statement starts with a seventeen-character Hollerith field, the first of the seventeen characters being a blank (note that blanks are counted here!). The computer therefore constructs the following "output record" within its main machine

memory:

| | | P | R | O | J | E | C | T | | M | O | O | N | B | E | A | M |

The next thing encountered in scanning the FORMAT statement is a *slash* "/". We have introduced this symbol already, in Chapter IV, as indicating the end of a printed line. More generally, *the slash indicates the end of one record, and the start of another record.* Thus nothing more goes into this particular output record, and we proceed to issue the print command to the line printer. The effect is to produce the following printed line:

| P | R | O | J | E | C | T | | M | O | O | N | B | E | A | M |

Note that the very first character of the output record for a line printer is omitted from the printed line. This peculiarity applies to printed output only (for example, a punched output card would contain all seventeen characters of the output record), and is related to a certain feature of full, rather than basic, FORTRAN IV. (*) *It is advisable to use a blank as the first character of every output record destined for the line printer.* In this way, compatibility with full FORTRAN IV is assured.

Having printed out the first line, we now start to construct another output record. The first field descriptor encountered after the slash is "10H REPORT NO". Hence the first thing in our new output record is a ten-character Hollerith field containing the characters " REPORT NO".

The next field descriptor is "I3", specifying conversion of an integer. We therefore proceed to look at the list of the WRITE statement, and in particular at the first item within that list, to wit: MONTH. This is indeed an integer identifier name (if it had not been so, the operating system would have issued an execution time error message, and stopped further program execution). We fetch the number stored in memory location MONTH, convert it from internal machine form to the form specified by "I3", to wit, a string of three characters consisting of two blanks and the numeric character "8". Since the next thing encountered in the FORMAT statement is another slash, this completes our output record, which now reads:

| | | R | E | P | O | R | T | | N | O | | | 8 |

This is a thirteen-character record (ten in the Hollerith field, three in the "I3" field). When this is now printed out on a line, the very first character is again suppressed, so that the second printed line contains the twelve characters:

| R | E | P | O | R | T | | N | O | | | 8 |

There are two slashes next. The first slash terminates the output record under construction, and results in output of the above printed line. The second slash tells us to output an *empty record.* That means an empty line for printed output, or a blank card for punched card output. Since we are sending our output record to the printer here, it means that we now produce one blank line on the printed output.

(*) In full FORTRAN IV, the first character of the output record is a "printer control character", that is, it represents a coded command to the printer, rather than a character actually to be printed. By use of the printer control character, it is possible, for example, to instruct the printer to skip to the start of a new page, so that the new line appears as the top line on a new page. This facility is not provided in Basic FORTRAN IV, where the first character of the output record is simply ignored altogether.

The fourth output record starts with an integer, from memory location NGIN, in "I5" form, then a ten-character Hollerith field.

The fifth, and last, output record starts with a fifteen-character Hollerith field, then a real number (from memory location BLNCE) in "F10.3" form, and finally another Hollerith field, with eleven characters in it. This fifth output record is the last one, since the next thing encountered in the FORMAT statement is the closing bracket, and since the entire list contained in the WRITE statement has been printed out by now. The full printed output reads:

```
P R O J E C T   M O O N B E A M
R E P O R T   N O     8

  5 2 6   E N G I N E E R S
B A N K   B A L A N C E   =          - 9 . 0 7 7     M E G A B U C K S
```

This is five lines of output altogether, the third line being blank. Note that we have lost some significant figures of the final number by specifying the field descriptor "F10.3", which gives us only three figures after the decimal point. The actual number -9.07742 was printed out as -9.077. The F-format suffers from this danger, as well as from the danger of attempting to print out a number which is so large (has so many digits to the left of the decimal point) that we overrun the entire field width. The E-format gets around both these deficiencies, but the price we pay for this is a much more awkward and hard-to-read form of printed output.

In our example, the FORMAT statement and the WRITE statement matched precisely, in the sense that there were exactly enough (three) numeric field descriptors for the three numbers to be printed. All the other field descriptors were for blank or for Hollerith fields. Such a precise match is not necessary. For example, consider the already used statements:

```
1 FORMAT (7E16.6)
  NPRNT=3
  WRITE (NPRNT,1) WAGE,HOURS,GROSS,TXDDC,PAY
```

Here the list contains five numbers to be printed, whereas the FORMAT statement specifies seven fields in the output record, each field with field descriptor "E16.6". What happens now is that we use the first five fields for the five numbers, to start with. In scanning the FORMAT statement, we then encounter a sixth field "E16.6", but this time, there is no more item in the list of the WRITE statement. This is our signal for closing the output record, and printing out a line with five numbers on it.

A related case is the "pure message" which we have used already in Chapter IV, that is, a WRITE statement with no list at all, and an associated FORMAT statement containing only blank fields and Hollerith fields and perhaps some slashes. In this case, the output record (or records) are assembled straight from the information within the FORMAT statement. No number is fetched from memory or converted. Note that the numbers, if any, appearing within a Hollerith field (such as "35" in "35 MAY 1984") are just some more characters, not really different from alphabetic or any other characters. Numbers within a Hollerith field are taken from that field within the FORMAT statement. They are not taken from some memory location used to store numbers.

There is also the possibility that the field descriptors within a FORMAT statement are fewer than is needed to cope with all the items in the list of the WRITE statement. This case is discussed later on in this Chapter, see page 78.

Section D: Fields and Field Descriptors for Input.

So far, we have discussed output of information, that is, the WRITE statement and its associated FORMAT.

In order to input information, we use the READ statement of the FORTRAN language:

READ (*n,m*) *List*

This command organizes reading of information from the input device with number *n* (this number may be given as an unsigned integer, or as an integer variable identifier name), according to the specifications contained in the FORMAT statement with the statement number *m*. Numbers are read in one by one, are converted to internal form, and are stored into successive memory locations taken one after another from the simple list *List*.

The FORMAT statement still contains field descriptors, as before, but this time they describe *input fields,* that is, groups of adjacent columns on an input punched card. The sum of all the field widths within one input record must not exceed the maximum number of columns which can be read from one card (this maximum is either 80 or 72 for input punched cards, and is 40 for input marked cards).

However, the field descriptors which we have already encountered for output fields, have slightly different meanings for input fields, and this is what we shall discuss now.

First of all, the blank field descriptor "*n*X" now specifies skipping over *n* characters of the input record (of the input card). The information in these *n* columns is *ignored.*

There is no change in the meaning of the field descriptor for integers, "*n*I*w*". This specifies *n* fields of width *w* each, each field containing an integer which must be right-adjusted within its field. In practice, this right adjustment is likely to cause errors, and thus the punching of input cards with integers on them must be done with exceedingly great care. If an integer is displaced from its correct position within its field, then it is interpreted as a different integer by the machine. For example, consider the number "365" in an "I5" field. If this is the first field on the card, then the correctly punched number has the digit "3" in the third column of the card, the digit "6" in column 4, and the digit "5" in column 5. Suppose now, however, that a mistake was made in punching the card, so that the number is displaced one over to the left:

		3	6	5			365 *mis*punched in "I5" field, interpreted as "3650"

The machine may simply reject this field altogether (this depends on the FORTRAN compiler used). But with most FORTRAN compilers, the blank in position five of the field is understood as a zero, so that the number read from the card is 3650 rather than 365.

If the integer is displaced to the right, by mistake, the situation is even worse. Let us take the FORMAT specification "2I5", assume that the two desired numbers are 365 and 12, respectively, and that the number 365 has been mispunched one column over to the right. The mispunched card looks like this:

			3	6	5			1	2	

The computer reads the first field as "36", and the second field as "50012" (that is, the blanks are replaced by zeros).

Thus, input of integers should be minimized. Whenever integer input is unavoidable, immediate echo-checking is mandatory, and so is immediate checking of the input numbers for "reasonableness". For example, it is most unlikely that the number 50012 is a reasonable input number for a program in which 12 is a reasonable input number. You should decide in advance what a "reasonable" range is for each of your input numbers, and you should test that each actual input number lies within its range, by means of an IF statement. If one input number fails the test, print out the faulty number, along with a message to say why it failed the test. But do not stop input at this point. Rather, continue to read all the input numbers, thereby going through the checking procedure for all the later numbers as well, in the hope of finding more mistakes. Do not, however, start the main computation which the program is designed to do: there is no point in calculating with unreasonable input numbers. Machine time costs money!

Fortunately, the field descriptors for real numbers can be used in a more relaxed fashion for input (rather than output) of real numbers.

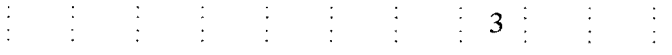
The field descriptor *n*F*w.d* specifies *n* fields of width *w* each, each field containing one real number in ordinary decimal form. However, unlike the situation in output, the input number need not be right-adjusted within its field, and it need not have exactly *d* decimal digits after the decimal point. In fact, the "*.d*" specification has no influence whatever as long as the number which is punched on the card has an explicit decimal point. That decimal point takes precedence over the "*.d*" specification. The "*.d*" is inspected and used only if the decimal point has been left off the punched number.

Our "standard" FORMAT specification for input of numbers has been:

```
2 FORMAT (4F10.0)
```

This specifies four fields, each of width ten, on the input card. The ".0" at the end makes no difference as long as each number is punched with a decimal point in it.

If the decimal point is omitted by mistake, then the specification ".0" informs the machine that it should assume a decimal point at the extreme right end of the field. That is, it should assume no (0) decimal digits after the decimal point, if the number is right-adjusted. This may be what the programmer wanted, but need not be so. For example, suppose that the programmer wanted to use the number "3.0", but has made the mistake of punching only the single digit "3", say in the eighth position within this field of width ten, as follows:



The number 3.0 mispunched (no decimal point!) in an F10.0 field.

The machine replaces the blanks in positions nine and ten of the field by zeros, and supplies a decimal point to the extreme right of the field. The result is the number "300.", rather than the desired number "3.". Conclusion: *Remember that decimal point!*

The possibility of placing the number anywhere within the field (not necessarily right-adjusted) is most welcome. The number should be placed roughly in the middle of the field, thereby minimizing the danger of overrunning the field boundaries on either side.

The field descriptor *n*E*w.d* also can be used more flexibly in input, than in output. In fact, the flexibility of this specification for input is enormous, and makes it the preferred input specification in many cases. First of all, it is not necessary to supply the number right-adjusted in its field; it can be, and should be, placed roughly into the

middle of the field; *however, if an exponent is present at all, this exponent must still be right-adjusted.* The mantissa need not have exactly *d* figures after its decimal point, but can be any decimal number, as long as the decimal point is supplied explicitly.(*) If the number of shifts is meant to be zero, it need not be supplied at all. If shifts are required, the code letter "E" may be omitted if there is an explicit sign ("+" or "-") on the number of shifts. The number of shifts need not be written with exactly two digits (as it is in output). But, unfortunately the actual number of shifts must still be right-adjusted in the field. This is a most regrettable limitation on what would otherwise be a very flexible, and hence very welcome, form of data specification for input numbers.

For example, the following are some of the many forms in which the number "429.6" can be input with the "E16.6" field descriptor. The mantissa can, and should, appear near the middle of the field, but in those cases where an exponent is used, that exponent must be right-adjusted:

429.6	+429.6	4.296 E02	4.296 E2
4.296 E+2	4.296 +2	+4.296 +2	4296.0E-01
4296. E-1	4296. -1	0.429600E 03	

The last of these many alternate forms is the one which would appear in output, with this field descriptor. In output, everything would be right-adjusted within the field, whereas in input, the mantissa may be centered in the field, only the exponent having to be right-adjusted.

Let us now show an example of input of numbers. Suppose that we wish to read a card containing numerical values of NPART and of PRICE. The first, the number of the part, is an integer; the second, the price, is a real number.

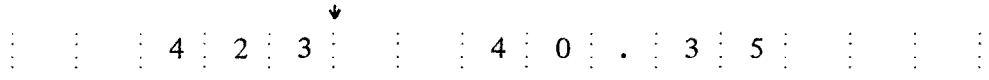
First we must decide what fields on the card to use, with what field widths. Since the integer must be right-adjusted in its field, it is advisable not to use too wide a field. It is easier to check on the adjustment if the field width is small. We shall choose a field of width five for the integer (this allows for up to 99999 different numbered parts), and a field of width ten for the real number (this is a fairly generally useful convention).

We shall assume that PRICE is never going to be so high that we need to go to E-format (the maximum price allowed is therefore 9999999.99 dollars; can you explain why?). Thus our field descriptors are "I5" and "F10.2", respectively. The ".2" in "F10.2" is irrelevant as long as the input number has its decimal point punched on the input card.

The resulting program segment is:

```
      READ (NREAD,1200) NPART,PRICE
1200  FORMAT (I5,F10.2)
```

If the part number is "423" and the price is $40.35, then the data card should be made up as follows:



We have indicated the boundary between the two fields by an arrow. Only the integer must be right-adjusted. The real number is centered in its field.

(*) It is even possible to omit the decimal point of the mantissa, whereupon the machine supplies an assumed decimal point in accordance with the ".d" specification. This practice is likely to cause errors, however, and is definitely not recommended. Include that decimal point!

There is nothing more to say about *input of numbers,* but there are occasions when we wish to *input character strings* rather than numbers. For example, in a payroll program, we would be well-advised to keep track of the *name* of each employee, along with his hourly wage and with the number of hours he worked during the week. The name should be on each data card, and should also appear on the printed output for the payroll. This can be done, but is a bit tricky, so we shall defer this matter to Section E (page 75).

For the moment, we shall consider a somewhat simpler problem, also related to the payroll program. It would be nice to be able to supply a first data card each new week, giving *heading information* for this week's payroll, for example:

```
SQUEAKYTOY COMPANY PAYROLL, WEEK STARTING 35 MAY 1984
```

If we can arrange to read cards of this type as data cards (rather than including the message as part of the FORTRAN program), and to use the data so read in to produce a heading line for the printed output of the payroll, then our payroll program has immediately become much more useful. The compilation of the program (translation from FORTRAN to machine language) can now be done once and for all. The translated program can be obtained on punched cards, punched by the card punch attached to the computer, by asking the FORTRAN compiler to produce an "object deck". This object deck can then be used for each week's run. It is now no longer necessary to change the FORTRAN program, and compile the changed program, every week, just to get a new heading message! On the contrary, we can change the heading message each week, merely by changing one data card.

It will be easier to understand what follows if we say something first about how the information supplied within a FORMAT statement appears after compilation, within the memory of the computer. Let us consider the FORMAT statement:

```
1001 FORMAT (1X,71H.....................................................
     1.....................)
```

During translation of this statement from FORTRAN to machine language, the statement number and the characters "FORMAT" are suppressed. But the information itself, that is, the two parentheses and everything contained between them, is stored away in the machine, as a coded character string, somewhere in a region of machine memory not reserved for storage of numbers. Thus, after translation from FORTRAN into machine language, the following (suitably coded) character string sits somewhere in the main memory of the computer:

```
(1X,71H.....................................................................)
```

We note, incidentally, that the character string stored away in the machine pays no attention to the fact that the original FORMAT statement had to be split between a first card and a continuation card.

Let us now ask what happens when this particular FORMAT statement is used with a READ statement, rather than with a WRITE statement. That is, we assume that the FORTRAN program contains the statement:

```
READ (NREAD,1001)
```

We further suppose that the data card with the message about the Squeakytoy Company is actually sitting in the card reader, ready to be input, at the moment this READ statement is reached during execution of the object program. (We remind you once more that reading of input data cards occurs during the execution phase, not during the compile phase).

The effect of the "1X" is to skip over the first column of the data card. This is the reason we left that column blank (see line 10 of the preceding page).

The effect of "71H" followed by the 71 dots is quite different, however. What happens is that the next seventy-one columns on the data card are read, and these seventy-one characters *replace* the seventy-one dots within the character string stored inside the machine. The character string inside the machine thus becomes:

```
(1X,71HSQUEAKYTOY COMPANY PAYROLL, WEEK STARTING 35 MAY 1984                )
```

From this moment on, whenever FORMAT statement number 1001 is mentioned in the program, this new character string is going to be used as the FORMAT information. For example, suppose that the next statement in the FORTRAN program is:

```
      WRITE (NPRNT,1001)
```

The computer looks at the FORMAT information stored in its memory, and therefore prints out a line as follows (the blank from "1X" is suppressed on the printed page):

```
SQUEAKYTOY COMPANY PAYROLL, WEEK STARTING 35 MAY 1984
```

We have now achieved what we set out to do: The message of the week can be supplied on a data card (rather than as part of the FORTRAN source language program), thereby making the payroll program more flexible.

Generally, whenever a field descriptor "*n*H", followed by exactly *n* characters of any sort (it does not matter what these *n* characters are), appears in a FORMAT statement for a READ operation, then the effect is that the next *n* characters are read from the data card currently in the card reader of the computer, and these *n* characters are used to replace the previous *n* characters in the FORMAT information stored inside the memory of the computer. From then on, any WRITE statement referring to this particular FORMAT uses the newly read-in characters for the output of that particular Hollerith field. (Note: It is possible to have input of numbers, and of character strings, from the same data card (different fields); but we advise strongly against this; errors are too likely).

Section E: Input and Output of Names in a List.

Basic FORTRAN IV, unlike the full FORTRAN IV, provides only very limited facilities for handling strings of characters, such as names. Nonetheless, some quite useful things can be done even in Basic FORTRAN IV, and we present an example of this in the present section. This material can be skipped by readers not interested in commercial data processing, or having the full FORTRAN language at their disposal.

In our payroll program, it would be most useful to have the *name* of each employee on the data card which gives his hourly wage and the number of hours he worked; and to have the same name appear on the payroll table, along with his payroll information. In this section, we show how this can be done.

We shall change the way our input cards are punched, as follows: the first column will be blank, the next 19 columns will be used for the name of the employee, and only then will we punch the values of WAGE and HOURS for this employee, each in a field of width 10. We have left the first column blank with malice aforethought, since we shall wish to use the same FORMAT statement with which the input card is read, as a FORMAT statement for printing an output line of the payroll table.

The FORMAT statement has to be planned with some care, since it is to serve two purposes at once (reading of input cards, writing of output lines). The first two field descriptors must be "1X" and "19H . . . . . ", respectively, to accommodate the blank first character and the subsequent name-field of width 19. The 19 dots will be replaced by the name, each time a new card is read. Next, we require field descriptors for the input numbers, but these field descriptors must do

double duty, since they will be used also for output numbers. The field descriptor "F10.2" for each such field is adequate for our purpose: it will result in reasonable output (a number of dollars and cents, with a decimal point between the dollars and the cents); it will also serve for input. For input, two such fields would suffice, since we need to read only two numbers from the card. But for output we shall need five such fields, since we want to print out five numbers (the values of WAGE, HOURS, GROSS, TXDDC, and PAY). Thus we shall use the field descriptor "5F10.2" within the FORMAT statement:

```
1501 FORMAT (1X,19H..................., 5F10.2)
```

We shall read the normal input cards with the statement:

```
1500 READ (NREAD,1501) WAGE,HOURS
```

and we shall print each line of the payroll table with the statement:

```
     WRITE (NPRNT,1501) WAGE,HOURS,GROSS,TXDDC,PAY
```

Compared with the program on page 42, the change is that both the READ and the WRITE statement refer to the *same* FORMAT statement, namely statement number 1501.

Let us suppose that a typical employee data card reads:

```
 BLOW, JOSEPH Q          2.25          35.0
```

where the first column has been left blank. The name appears in a field extending from column 2 to column 20 on the card. The value of WAGE appears in the field extending from column 21 to column 30 of the card. The value of HOURS appears in the field extending from column 31 to column 40 of the card.

As a result of execution of the READ statement (number 1500), the following things happen: The FORMAT information stored inside the machine is modified to read:

```
(1X,19HBLOW, JOSEPH Q       , 5F10.2)
```

Furthermore, the value of WAGE is set equal to 2.25, the value of HOURS to 35.0.

We then proceed with the program to calculate GROSS, TXDDC, and PAY, which turn out to be 78.75, 9.45, and 69.30, respectively. Thereafter we reach the output statement WRITE, which now produces the following output printed line:

```
BLOW, JOSEPH Q            2.25       35.00      78.75      9.45      69.30
```

This is quite an improvement over the earlier form of output, see page 43.

We now present, first, the revised FORTRAN program in full, with all the necessary changes (this includes a change in the FORMAT statement, number 1100, which is used for the table heading: since the table looks different, the table heading must be changed, also). Then we shall present the data cards which are to be used. Finally, we reproduce the appearance of the execution time printed output.

First, then, the revised FORTRAN program (compare with page 43):

```
C      PAYROLL PROGRAM
       NREAD=1
       NPRNT=3
C      CHECK THE ABOVE NUMBERS FOR YOUR INSTALLATION
C      READ A DATA CARD WITH A HEADING MESSAGE, THEN PRINT OUT MESSAGE
 1000 READ (NREAD,1001)
 1001 FORMAT (1X,71H.....................................................
     1.....................)
 1010 WRITE (NPRNT,1001)
C      NOW PRINT THE TABLE HEADING, SUITABLY MODIFIED
       WRITE (NPRNT,1100)
 1100 FORMAT (/9H EMPLOYEE,17X,4HWAGE,5X,5HHOURS,5X,5HGROSS,6X,3HTAX,7X
     1,3HPAY/)
C      LOOP STARTS HERE
 1500 READ (NREAD,1501) WAGE,HOURS
 1501 FORMAT (1X,19H.....................,5F10.2)
```

```
      IF(WAGE) 3000,3000,1700
 1700 IF((WAGE - 1.50)*(4.25 - WAGE)) 1800,1900,1900
 1800 WRITE (NPRNT,1810)
 1810 FORMAT (48H ***** UNREASONABLE VALUE OF WAGE ON LINE BELOW    )
 1900 IF(HOURS*(40.0 - HOURS)) 1950,2000,2000
 1950 WRITE (NPRNT,1960)
 1960 FORMAT (49H ***** UNREASONABLE VALUE OF HOURS ON LINE BELOW    )
 2000 GROSS=WAGE*HOURS
      TXDDC=0.12*GROSS
      PAY=GROSS - TXDDC
      WRITE (NPRNT,1501) WAGE,HOURS,GROSS,TXDDC,PAY
      GO TO 1500
 3000 STOP
      END
```

Note that we have omitted the earlier "standard" FORMAT statements from this program, since they are no longer used. Note also that most of the FORTRAN program is devoted to input of data, checking of the input data, and organization of neat, easily legible, printed output. Only three statements (starting with statement number 2000) are devoted to the actual computation! This is typical of data processing programs, less true of scientific and engineering programs. But even in scientific and engineering work, careful handling of input and output is essential, and takes up considerable space in a well-designed program.

Next, we shall show the contents of the data cards which we shall use for this program. The first data card contains the heading line. This was discussed on pages 74 and 75. This data card is read by the READ statement with statement number 1000 in our program.

All subsequent data cards are read by the READ statement with statement number 1500. They are employee data cards, including two cards with deliberately bad data. All these have column one left blank, the name starting in column two. The last, blank card results in transfer of control to statement number 3000 in the program, from the "IF(WAGE) 3000,3000,1700" statement.

```
SQUEAKYTOY COMPANY PAYROLL, WEEK STARTING 35 MAY 1984
BLOW, JOSEPH Q          2.25      35.0
FOX, BRER               2.75      26.0
BLIMP, COL. H. Q.       3.20       0.0
RICH, RICHIE            9.99      34.0
SACK, SAD               2.25      - 7.5
OVER, LEFTIE            2.50      35.5
blank                   blank     blank
```

Finally, we show the printed output from this program with these data cards (execution time output only, exclusive of the listing of the FORTRAN program):

```
SQUEAKYTOY COMPANY PAYROLL, WEEK STARTING 35 MAY 1984

EMPLOYEE                WAGE     HOURS     GROSS     TAX       PAY

BLOW, JOSEPH Q          2.25     35.00     78.75      9.45     69.30
FOX, BRER               2.75     26.00     71.50      8.58     62.92
BLIMP, COL. H. Q.       3.20      0.00      0.00      0.00      0.00
***** UNREASONABLE VALUE OF WAGE ON LINE BELOW
RICH, RICHIE            9.99     34.00    339.66     40.76    298.90
***** UNREASONABLE VALUE OF HOURS ON LINE BELOW
SACK, SAD               2.25     - 7.50   - 16.88    - 2.02   -14.85
OVER, LEFTIE            2.50     35.50     88.75     10.65     78.10
```

This output for the payroll table should be compared with the output shown on page 43, near the bottom of the page. The advantages of using a better FORMAT, plus keeping track of employee names, plus providing a variable heading line for each new week, should be apparent.

Note that the program does not stop altogether when a bad data card is encountered. This allows us to find further errors, as well as getting output for Mr. Leftie Over from his (correct) data card.

Section F: Repeated Groups of FORMAT Fields, Repeated Scanning of a FORMAT statement.

The FORTRAN language provides several methods designed to make FORMAT statements shorter, and easier to write. The first method, the "duplication factor" before an individual field descriptor, has been used already.

In addition to this, groups of field descriptors can be repeated, as illustrated below:

```
5500 FORMAT (3(I5,F10.2),I10)
```

is equivalent to

```
5500 FORMAT (I5,F10.2,I5,F10.2,I5,F10.2,I10)
```

This device is "limited" in the sense that it is not possible to have bracketed groups of field descriptors within another such group. But several bracketed groups may appear next to each other in a FORMAT statement. For example, the statement

```
5501 FORMAT (3(I5,F9.2),4(I7,F9.3))
```

is permissible, and is equivalent to

```
5501 FORMAT (I5,F9.2,I5,F9.2,I5,F9.2,I7,F9.3,I7,F9.3,I7,F9.3,I7,F9.3)
```

The advantage gained from the use of bracketed groups of field descriptors is apparent.

A different form of "repetition" occurs when the FORMAT statement contains fewer field descriptors for numbers, than there are numbers to be printed or read. For example, consider the following two statements:

```
      READ (NREAD,2) A,B,C,D,E,F,G,H,P,Q
    2 FORMAT (4F10.0)
```

The FORMAT statement specifies four fields of width 10 each. Thus we read four numbers from the first data card, and store these four numbers into locations A, B, C, and D, respectively. We are now at the end of the FORMAT statement. But when we look at the list of the READ statement, we see that there are more numbers to be read in. What we do in such a case is to place the next data card into the reader, and read four numbers off that card. The four numbers on the second data card are placed into storage locations E, F, G, and H, respectively. Since the list is still not exhausted, we proceed to read the third data card. Only the first two fields of that data card are input, and the numbers in those fields are placed into memory locations P and Q, respectively.

The same rescanning of a FORMAT statement occurs in a WRITE operation. For example, the statements

```
      WRITE (NPRNT,1) A,B,C,D,E,F,G,H,P,Q
    1 FORMAT (7E16.6)
```

result in two lines of output. The first line contains the values of A, B, C, D, E, F, and G; the second line contains the values of H, P, and Q.

The reexamination of a FORMAT statement in the case of a too long list does not necessarily include the entire FORMAT statement. Rather, *if the FORMAT statement contains bracketed groups of FORMAT codes, then the reexamination commences at the right-most such group.* Earlier parts of the FORMAT statement are ignored. If there are no bracketed groups of FORMAT codes, on the other hand, then the entire FORMAT statement is reexamined each time.

This feature can be used to include heading information for a table of numbers within the same FORMAT statement which controls output of the numbers themselves. For example, suppose we wish to print out a table containing the values of A1 and B1 on the first line, A2 and B2 on the second line, and A3 and B3 on the third line. We also want to output a table heading. This can be done as follows:

```
      WRITE (NPRNT,5503) A1,B1,A2,B2,A3,B3
 5503 FORMAT (35H VALUES OF A AND B FOR THREE CASES  /(2E16.6))
```

The effect of this is to precede the table of numbers by a line with the table heading. This table heading is not reexamined afterwards; only the right-most group "(2E16.6)" is looked at again.

Summary and Formal Definitions for Chapter VII.

1. *External Record:*

An external record consists of a number of *characters.* Usually, a number of adjacent characters within such a record are combined into a *field,* so that the record may be said to consist of fields of characters. The number of characters in a field is called the *field width.* The sum of all the field widths in a record is the *record size.* This must not exceed the *maximum record size* for the input/output device in question (80 characters for a punched card, 40 characters for a marked card, 121 characters for a printed line on a line printer with 120-character line width). In line printing, the first character of the output record is not printed.

2. *Field Separators:*

The comma "," and the slash "/" are field separators within a FORMAT statement. The slash indicates also the termination of one external record, and the start of the next record. In reading, a slash means "feed in the next data card"; in writing, a slash means "start a new printed line".

3. *Field Descriptors:*

*n*X specifies a blank field of width *n* for output; it specifies skipping *n* characters of the data card for input.

*n*H... (the dots indicate exactly *n* characters after the "H", blanks being counted here) specifies a Hollerith field of width *n*. In output, this set of characters is printed out. In input, the next *n* characters are read from the data card, and are placed into the format information stored in the machine.

*n*I*w* specifies *n* fields, each of width *w,* each containing one integer; each integer is right-adjusted in its field.

*n*F*w.d* specifies *n* fields, each of width *w,* each containing a real number in basic real number form. In output, each number is printed with *d* digits after the decimal point, right-adjusted in its field. In input, the number need not be right-adjusted, and the position of the actual decimal point overrides the "*.d*" specification.

*n*E*w.d* specifies *n* fields, each of width *w,* each containing a real number in E-format (mantissa followed by a number of shifts). In output, the mantissa has a "0" before the decimal point of the mantissa, *d* digits after the decimal point, then the letter "E", then the number of shifts as an unsigned or negative integer with two digits. The field width *w* must be at least equal to *d*+7. In input, the number need not be right-adjusted in the field, but if an exponent is present at all, then this exponent *must* be right-adjusted.

4. *Bracketed Group:*

n(...) where the parentheses contain field descriptors separated by field separators, specifies the contents of the brackets *n* times over. Brackets within brackets are not permitted.

5. *FORMAT statement:*

m FORMAT (*s g s' g' s" g" ... s*)

where *m* is a statement number. The first and last *s* are either absent or consist of any number of slashes. *g, g', g", etc.* are either field descriptors or bracketed groups. *s', s",* etc. are field separators.

In execution, the format information within the parentheses is combined with the input/output list of the READ/WRITE statement being executed. The format information is read from left to right. If and when a field descriptor for a numeric field (integer or real) is encountered, the next item on the list is transmitted. Transmission stops when either: (i) a numeric field descriptor is encountered but the list is exhausted, or (ii) the closing right bracket in the FORMAT statement is encountered and the list is exhausted. If the closing right bracket is encountered before the list has been exhausted, the FORMAT statement is re-scanned, starting with the right-most bracketed group (if there are such groups) or with the beginning of the format information (if there are no bracketed groups).

DRILL EXERCISES FOR CHAPTER VII.

1. In a WRITE statement, how do we specify: (a) what is to be written, (b) what device is to be used for the output, and (c) the way in which the output is to be written?

2. Find the FORTRAN language errors in the following WRITE statements (assume that the variable NPRNT has been set appropriately, earlier in the program).
   a) WRITE (NPRNT,1001,) DA,D,DY
   b) WRITE (NPRNT,1002),QUO,VADIS
   c) WRITE (NPRNT,1003) WEE,WILLY,
   d) WRITE (NPRNT,1004) 32.8,26.3,143.92
   e) WRITE (NPRNT,1005) A,B**2,C

3. Write FORMAT statements to do the following:
   a) 10 integers to a line of line-length 120 characters.
   b) print an integer in a field of width 10, then use the rest of a 120-character line for 5 real numbers in E-format, each with 6 significant digits in the mantissa.
   c) input cards are prepared with the following layout: 2 integers, each in a field of width 5, followed by 3 ordinary decimal numbers, each in a field of width 10. We wish to read these cards.
   d) read input punched cards containing: an integer in a field of width 5, then a real number in a field of width 15, then another integer in a field of width 5, then a real number which may be in E-format in a field of width 15, then 4 integers each in a field of width 5, finally a real number in ordinary decimal form, in a field of width 10.

4. Check the following FORMAT statements for errors:
   a) 1009 FORMAT (36I12)
   b) 1010 FORMAT,(9I12/9I12/9I12/9I12)
   c) 1011 FORMAT,5E20.7
   d) 1012 FORMAT,5E20.7)
   e) 1013 FORMAT (4F10.9)
   f) 1014 FORMAT (4E12.6)

5. Write the FORMAT statement to accompany the WRITE statement

   WRITE (NPRNT,1016) K,A,B,C,D,E,F,G

   if it is desired to have the output appear as follows:
   a) the integer K in a field of width 15, followed by A, B, and C in E-format, to 6 significant figures, each within a field of width 25; then start a new line containing the remaining numbers in ordinary decimal form, with three figures after the decimal point; the field for D should be right underneath the field for A, the field for E underneath the field for B, the field for F underneath the field for C.
   b) the output starts with a line reading NEW ITERATION, then a blank line, then a line with the message K= followed by the value of K; then another blank line, then a line with the values of A, B, and C, each preceded by an appropriate message, then another blank line, then the values of D, E, and F each preceded by an appropriate message; then a line containing the value of G preceded by a message, and finally one blank line. The appearance of the numbers themselves should be as in (a); proper alignment is desired.

6. A program produces a table of monthly repayments on a loan, each line of the table being produced by:

   2000 WRITE (NPRNT,2001) MONTH,DEBT,CHARG,PAYMT,REDUC,RMAIN
   2001 FORMAT (I8,5F12.2)

   a) Describe in detail the appearance of each printed line;
   b) Produce a WRITE statement and FORMAT statement for a suitable table heading.

7. Improve the program of drill exercise 11 of Chapter IV (page 46) to compute and print out a value of the integer MONTH for each repayment, this being the number of the repayment; use the results of the present exercise 6.

8. Produce and print out a table of numbers, their squares, cubes, and fourth powers, for the integers 1 to 10, inclusive, with a proper table heading.

PROGRAMMING EXERCISES FOR CHAPTER VII.

1. Revise drill exercise 8 above by reading in three integers K, L, and M, such that the first line of the table is for integer value K (rather than for 1), successive lines of the table refer to values differing from each other by an amount M (e.g., line 2 is for K+M, line 3 is for K+2*M, line 4 for K+3*M, and so on), and the table output terminates when the next value to be used would exceed the value of L. (For example, if the input numbers are K=3, L=8, and M=2, then the table contains the integer values 3, 5, and 7; the next value, 9, would already exceed L=8).

2. Improve the program above by (a) echo-checking the input numbers, with an appropriate message, (b) testing the input numbers to make sure that L is greater than or equal to K, and that M is positive (not zero or negative); an appropriate message to be printed out if either condition is violated, and (c) making the program "recycling": after one table has been completed, we go back to read new values of K, L, and M, and produce another table. The process continues until we encounter a blank card; no error message is to be printed out for a blank card; rather, program execution is terminated.

3. Take any textbook of College Algebra and read up the method known as "Euclid's algorithm" for finding the greatest common divisor of two integers K and L. Then write a program which reads K and L as input data, echo-checks them (with an appropriate message), computes the greatest common divisor, and prints that out, again with a message. We return to read the next set of values of K and L, and continue in that way until we see a blank card.

4. Produce a table of square roots, cube roots, and fourth roots of the integers from 1 to 15, inclusive. Each line of the table should contain the value of the integer I, and the values of the roots, as ordinary decimal numbers with six places after the decimal point. A table heading should also be printed. (Note: What is wrong with the statement CUBRT=I**(1/3), and with the statement CUBRT=I**0.3333333 ?)

5. Alter program 4 to read input data K, L, and M, controlling the range of numbers appearing in the table in the same way as in programming exercise 1. Carry out the same tests as in 2(b), but also test that K is not negative (otherwise the square root and fourth root are not real numbers).

6. Find all prime integers between 3 and 120 and print them out one by one.
Hints: (1) It is sufficient to test the odd integers, (2) to test an odd integer I, divide it by all odd integers J between J=3 and J=I-2, inclusive, in each case testing whether the division "goes". The well-formed arithmetic expression I-(I/J)*J gives the value zero if and only if J divides I exactly (see page 59). If none of these divisions "goes", then I is a prime and should be printed out.

7. Use the power series

   $\sin(x) = x - x^3/(1*2*3) + x^5/(1*2*3*4*5) - x^7/(1*2*3*4*5*6*7) + \ldots$

   to find the sine function of a number x read in from a data card. Assume that x lies between -2 and +2, and stop computation when encountering a number x outside that range. The value of sin(x) is wanted with four correct figures after the decimal point.
   Hints: Organize a loop on the number N of the term in the series. Precede the loop with N=1, TERM=X, and SUM=TERM, thereby setting initial values of N, of TERM, and of SUM. The first statement within the loop is N=N+1, advancing the order number of the term to be computed. The new term is obtained from the previous term by the two statements

   ```
   DENOM=(2*N-2)*(2*N-1)
   TERM=-TERM*X**2/DENOM
   ```

   (explain why two separate statements are necessary!). We then update the SUM by

   ```
   SUM=SUM+TERM
   ```

   Next we check whether the last term (which has just been added in) is less than 0.00005 in absolute value. If it is, then the first four figures after the decimal point are correct (further terms of the series would make even smaller contributions, for this particular series), and we leave the loop.

8. In accounting, the use of floating point (real) numbers has the disadvantage that the round-off errors may make the totals add up incorrectly, by a few cents. In order to avoid this, business calculations are often done purely with integers, representing numbers of cents. Addition,

subtraction, and multiplication are then exact. To get a properly rounded (rather than a truncated) division, say by 100 for a percentage calculation, we must add half the divisor (i.e., the number 50) to the dividend, before doing the division. If MONEY is the amount and IRATE is the interest rate, expressed as a whole number of percent, then the FORTRAN statement is:

```
INTST=(MONEY*IRATE+50)/100
```

If the interest rate is expressed as a whole number of tenths of a percent (of pro mille), we write:

```
INTST=(MONEY*IRATE+500)/1000
```

In each case, the number added inside the brackets is half the divisor.

a) Explain the operation of these statements, in terms of FORTRAN integer arithmetic. Explain why they *fail* to work if one, but not both, of MONEY and IRATE are negative numbers.

b) Reprogram the debt repayment problem (drill exercises 6 and 7 on page 80) to use integer arithmetic only, printing out all amounts in cents, as integers. Make sure to program in such a way that all totals balance precisely.

c) In order to get a neater table printout, we want to print out each number of cents converted to a number of dollars and fractional dollars, for example, 35.63 for 35 dollars and 63 cents. But the internal calculations are still to be done in whole cents. If MONEY is the amount of cents, then the following conversion can be used to produce dollars in DOLLR (why?):

```
DOLLR=MONEY
DOLLR=0.01*DOLLR
```

Use this method to get a neater table printout from the calculation of part (b).

9. Make the changes, described in problem 8 above for the debt repayment program, for the payroll program given on pages 76 and 77.

10. In business reporting to executives of a large corporation, it is desired to round all quantities of money to the nearest thousand dollars. Given money amounts expressed as real numbers, how would you convert them to properly rounded numbers of thousands of dollars? Reprogram the debt repayment problem of drill exercises 10 and 11 on page 46, converting all quantities in the output table to the nearest thousand dollars (making sure, however, that the quantities in the table itself balance properly). Assume that the initial debt was between one million and fifty million dollars. Explain why integer arithmetic is *not* required for the internal calculation done within the program, before conversion to rounded form.

CHAPTER VIII

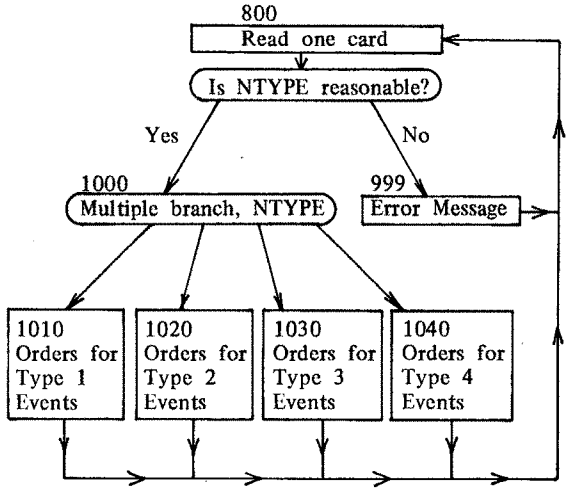
PROGRAM FLOW CONTROL, CONTINUED

Section A: The Computed GO TO.

A typical accounting system allows for a number of different types of business "events", for example, type 1 may be "sale of merchandise for cash", type 2 "sale of merchandise on account", etc. One input card is prepared for each event. The type of the event is in an I5 field, which is followed by up to four numbers I, J, K, and L, each in an I10 field, giving relevant information about the particular transaction (e.g., prices expressed in cents, part identification numbers, etc.). We want to read each card, then branch to different sections of the program to handle the accounting for different types of events. To simplify matters, we allow only four types here (the same principles apply to any number of event types). We shall branch to statement 1010 for events of type 1, 1020 for events of type 2, 1030 for events of type 3, and to 1040 for events of type 4 (the actual statement numbers can be chosen at will, but we recommend using a simple rule like the one just given). This multiple choice branching can be programmed by a (rather awkward) sequence of IF statements. But FORTRAN has a better method:

```
GO TO (1010,1020,1030,1040),NTYPE
```

The brackets contain a simple list of statement numbers. We go to the *first* of these (statement number 1010) if NTYPE equals 1; to the *second* if NTYPE equals 2; and so on. There is a *comma after the right bracket* (watch that one: a copious source of programming errors!), followed by the name of the indicator variable (NTYPE in our case). This "computed GO TO" fails if NTYPE is less than 1, or greater than the number of items in the list of statement numbers (greater than 4 in our case). Thus our program must test that NTYPE lies between 1 and 4, inclusive, and produce an error message if it does not.




```
      NREAD=1
      NPRNT=3
  800 READ (NREAD,801) NTYPE,I,J,K,L
  801 FORMAT (15,4I10)
      IF((NTYPE-1)*(4-NTYPE)) 999,1000,1000
C     CRAZY EVENT TYPE NUMBER
  999 WRITE (NPRNT,998) NTYPE,I,J,K,L
  998 FORMAT (11H WRONG CARD,I5,4I10)
      GO TO 800
 1000 GO TO (1010,1020,1030,1040),NTYPE
 1010 ... (accounting for type 1 events)
      GO TO 800
 1020 ... (accounting for type 2 events)
      GO TO 800
 1030 ... (accounting for type 3 events)
      GO TO 800
 1040 ... (accounting for type 4 events)
      GO TO 800
```

A single FORTRAN statement, the computed GO TO, is used to organize this four-way branching of control. Further branches can be inserted as needed. Note that the error message, starting with statement number 999, not only states that the card was bad, but gives the card image of that bad input card, exactly as that card was read in. This is most desirable. There is nothing more annoying in data processing than a message of the type "An error has occurred in a READ or WRITE operation." When you have been trying to input a deck of 2000 data cards, or to read three magnetic tapes, such a message is an unmitigated insult. Some deplorable operating systems do just that; but do not do the same thing to yourself, in your own program!

Section B: The Loop Control Statement "DO".

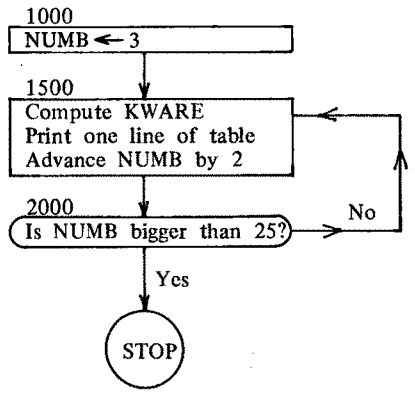
Let us consider the following simple problem: We want to produce a table of integers and their squares, for all the odd integers between 3 and 25, inclusive, i.e., for the numbers: 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, and 25.

As usual, we start by drawing a flow diagram. And, again as usual, we first concentrate on what must be done inside the loop; only afterwards do we decide what must be done before the loop is entered, and after the loop is completed.

Within the loop, we have some odd number NUMB, say. We must compute its square and store it into some memory location, call it KWARE. And we must output a line containing the values of NUMB and KWARE. Then we must increase the value of NUMB by 2, so as to get the next odd number, and we must test whether this next value of NUMB exceeds our upper limit, 25. If not, we go back for the next turn around the loop. If NUMB does exceed 25, the loop is finished.

All we need to do before entering the loop is to set NUMB equal to its initial value 3. All that is required after completing the loop is to stop program execution. (In general, it would be better to produce a table heading before entering the loop, but we want to restrict ourselves to absolute essentials for this example).

The flow diagram follows, and next to it, the FORTRAN program:




```
C       TABLE OF SQUARES
        NPRNT=3
 1000 NUMB=3
C       LOOP STARTS HERE
 1500 KWARE=NUMB**2
        WRITE (NPRNT,1600) NUMB,KWARE
 1600 FORMAT (2I8)
 1800 NUMB=NUMB+2
 2000 IF(NUMB-25) 1500,1500,2500
C       LOOP FINISHED
 2500 STOP
        END
```

You would do well to check through this program to make sure that the first line of the execution time printed output contains the numbers "3" and "9", the next line has the numbers "5" and "25", and so on, till the last line with the numbers "25" and "625". After that last line has been printed, NUMB is advanced from 25 to 27, by statement number 1800. The IF statement, number 2000, now sends us to statement number 2500, so that we stop program execution at this stage.

Loops of this sort, where an integer variable is advanced by the same amount each time, are so common in programming that FORTRAN IV provides a special statement, called the DO statement, to simplify the job of the programmer. We shall now show a FORTRAN program which does the identical thing as the one before, by use of the DO statement. It reads:

```
C       TABLE OF SQUARES
        NPRNT=3
 1000 DO 2000 NUMB=3,25,2
 1500 KWARE=NUMB**2
        WRITE (NPRNT,1600) NUMB,KWARE
 1600 FORMAT (2I8)
 2000 CONTINUE
C       WE GET HERE AFTER THE DO LOOP IS COMPLETED
 2500 STOP
        END
```

We note that the DO statement provides the following information:

1. The *end of the range of the loop,* i.e., the statement number of the last statement still within the loop; in our case the range of the loop extends from statement number 1500 to statement number 2000, inclusive. Note that the DO statement itself, statement number 1000, is *not* counted as being within the range of the loop;
2. The identifier *name of the loop variable,* in our case, NUMB;
3. The *initial value* of the loop variable, in our case, 3;
4. The *termination test value* of the loop variable, in our case, 25;
5. The *increment* of the loop variable, in our case, 2.

The last statement of this loop is statement number 2000, which is the dummy statement CONTINUE. This does not by itself give rise to any calculations, but serves as a convenient "no action" statement for closing the range of the loop. The last statement of a DO loop need not be a CONTINUE statement; in fact, it may be any executable statement other than one of: GO TO, IF, RETURN, STOP, PAUSE, DO. However, we recommend closing every DO loop with its own separate CONTINUE statement. This defines the range of each DO loop clearly, and also gives notice to the reader that at this point some DO loop has come to the end of its range.

Note: It is important to pay detailed attention to the standard form of the DO statement, in particular, the places where commas do, and do not, appear. There are no commas before or after the statement number; blanks in these places are necessary with many FORTRAN compilers, desirable with all of them. The name of the loop variable is followed by an = sign (again no commas). But there are commas between the initial value and the termination test value, and between the termination test value and the increment. These rules must be learned and obeyed.

The number of times that a DO loop is traversed before the loop is "satisfied" depends on the values of the parameters k, l, and m specified in the DO statement of type

DO n $J=k,l,m$

For example, with $k=3$, $l=25$, and $m=2$ (the values for our little loop before), the ratio

$$(l-k)/m = (25-3)/2 = 22/2 = 11$$

turns out to be one less than the number of times the loop is traversed (twelve times,

for the twelve odd integers 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, and 25). More generally the number of times around a DO loop with l greater than or equal to k is given by

Number of times around DO loop = 1 + Integer part of $(l-k)/m$

In particular, for $l=k$ (termination test value equal to initial value) the DO loop is traversed exactly once.

Naturally, one should not specify the termination test value to be less than the initial value. This makes no sense, since the loop variable is *increased* each time around the loop. FORTRAN compilers differ in how they handle this situation (which can happen by mistake). Most FORTRAN compilers produce programs which traverse every DO loop at least once. This is a dangerous trap to beginners, who imagine that a loop such as

```
DO 1200 JOKE=200,180,2
```

should not be traversed at all. In fact, it will be traversed once (for most FORTRAN compilers), even though the very first value of JOKE, JOKE=200, is already larger than the termination test value, 180.

In our little example before, we started coding the loop with the help of an IF statement, and showed the DO loop coding only afterwards. *Every* DO loop in FORTRAN can be recoded, if desired, by use of the IF statement for the termination test. Thus the DO statement is only a convenience, not a necessity.

The opposite is *not* true. There are loops which can be coded quite easily by use of the IF statement, but can not be coded at all with the DO statement. An example of such a loop was given a long time ago already, in Chapter III, page 29, for the calculation of a square root. We recall that each time around the loop we calculated a quantity called RATIO, and we kept going around the loop until the absolute value of RATIO became small enough to satisfy us. Such a loop can not be written as a DO loop, since it is impossible to predict just how many times we shall need to go around that loop. The decision to continue looping around, or not, depends on values currently computed within the loop.

Nonetheless, the two techniques can often be combined to good advantage. Let us consider the following example:

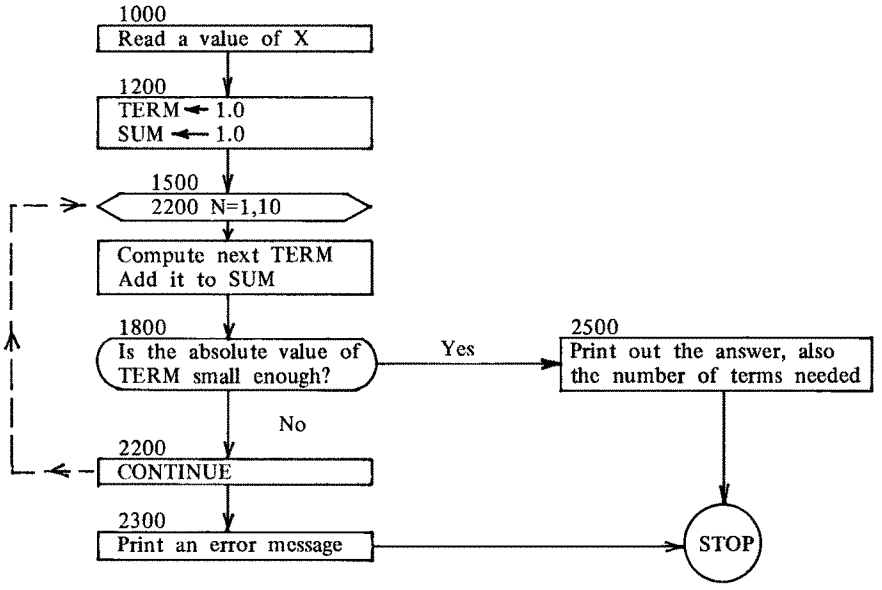
Problem: Given a value of X, read in from a data card, we want to find e^{-X} from the power series:

$$1 - x/1 + x^2/(1*2) - x^3/(1*2*3) + x^4/(1*2*3*4) - \ldots$$

Solution: We keep evaluating term after term of this series, and adding it to SUM, until either the last term added in is small enough (we shall say, smaller than 1.E-6), or until we have already evaluated a large number of terms of the series and the terms are still not small enough (we shall stop after ten terms here). In the latter case, we wish to print out a message that the series is unsuitable for this value of X.

In programming this solution, a beginner may be tempted to evaluate each new term of the series afresh, without making use of work done before. This is most inefficient, since in fact there is a simple relationship between any one term and the term before it in the series. Let us number the terms in the series in order: term number "0" is equal to 1.0, term number "1" is equal to -x/1, term number "2" is equal to $+x^2/2$, term number "3" is equal to $-x^3/6$, and so on. We obtain term number "N" from the previous term, by multiplying that previous term with -x/N.

The flow diagram below is based on this idea. The "squashed hexagon" in this flow diagram is meant to represent a DO statement. The parameters of the DO statement are written within the hexagon. They are: the statement number (2200) of the end of the range; the name (N) of the loop variable; the loop parameters k, l, and m (in our case, k=1, l=10, and m can be omitted because we want unit increment, m=1; FORTRAN permits omission of m if the value desired is m=1).



We observe that we can leave the DO loop of this program in two different ways: (a) The test at statement 1800 may succeed, and send us to statement number 2500; or (b) the DO loop may continue all the way to N=10, without success; in that case, we go on to statement 2300, where we print an error message. We have therefore succeeded in combining the DO loop and the IF techniques, within the same loop. The program reads:

```
C       EXPONENTIAL FUNCTION OF -X
        NREAD=1
        NPRNT=3
 1000 READ (NREAD,1010) X
 1010 FORMAT (E10.0)
 1200 TERM=1.0
        SUM=1.0
 1500 DO 2200 N=1,10
        EN=N
C       THIS HAS CONVERTED N TO REAL NUMBER FORM EN
        TERM=-TERM*X/EN
        SUM=SUM+TERM
 1800 IF(TERM**2-1.0E-12) 2500,2200,2200
 2200 CONTINUE
 2300 WRITE (NPRNT,2310) X
 2310 FORMAT (25H ITERATION FAILED FOR X = ,E15.7)
        GO TO 3000
 2500 WRITE (NPRNT,2510) X,SUM,N
 2510 FORMAT (7H FOR X=,E16.7,11H    EXP(-X)=,E16.7,8H    AFTER,I3,
     1 6H TERMS)
 3000 STOP
        END
```

When this program is run with 0.1 as the input number, we leave the loop with N equal to 5. But when we use 2.0 as the input number, we reach the failure exit, statement 2300, instead.

Let us now return to the general discussion of DO loops in FORTRAN.

When we jump out of a DO loop before the loop variable has reached the termination test value, then the value of the loop variable (N in our case) is available for use and printout. On the other hand, *after a DO loop has run to normal completion, the value of the loop variable is no longer available in its memory location.* For example, we are not allowed to assume that N has the value 10 when we reach statement number 2300. At this point, memory location N may contain anything at all, and should not be used before resetting it. This property of the DO loop in FORTRAN is a nuisance, and a copious source of programming errors, but you just have to learn to live with it.

Although it is possible to jump out of the range of a DO loop, FORTRAN expressly forbids jumping *into* that range, from a point outside the range. For example, if we were to replace statement number 3000 by

```
3000 GO TO 1800
```

then the FORTRAN compiler would detect, and diagnose, an error of FORTRAN grammar: jumping into the range of a DO loop, from a point outside that loop.

Note, however, that GO TO 1500 would be permissible: the DO statement itself is *not* considered part of the range of the DO loop.

The DO statement sets up a loop, and demands that the loop variable go systematically through the values k, $k+m$, $k+2*m$, $k+3*m$, and so on, until the termination test value l would be exceeded the next time round. FORTRAN forbids doing anything which might interfere with this normal operation of the loop (other than jumping out of the loop altogether). In particular, we are not allowed to reset the loop variable within the range of the loop, either by an assignment statement or by a READ statement. For example, the following statement must not be inserted just after statement 1500:

```
1600 N=N+3
```

This would be an attempt to advance the value of N by an additional three units, so that it would advance a total of four units each time around the loop. The DO loop jealously guards its right to control its own loop variable, and does not permit such crass interference. If we did want to advance N by four units each time around the loop, the way to do so would be by altering the DO statement itself, to:

```
1500 DO 2200 N=1,10,4
```

Similarly, if one or more of k, l, and m are given as integer variable names (rather than as unsigned integer constants), then it is forbidden to reset the values of any of these variables within the range of the loop. For example, within a DO loop such as

```
DO 2200 KOUNT=K,L
```

not only is it forbidden to alter KOUNT within the range of the loop, but also to alter the values of K or L.

Finally, we call attention to the restriction that k, l, and m may be unsigned integer constants or integer variable names, but not arithmetic expressions of more complex type. For example, the following is ungrammatical FORTRAN:

```
DO 2200 KOUNT=K,L+2
```

If we want to go from K to L+2, the way to do it is:

```
LPLS2=L+2
DO 2200 KOUNT=K,LPLS2
```

Let us summarize the restrictions and cautions for DO loops in FORTRAN:

Restrictions:

1. The parameters *k, l,* and *m* must all be positive (not zero or negative);
2. *l* must be greater than or equal to *k*;
3. It is forbidden to jump into the range of the loop from outside that range;
4. Neither the loop variable, nor any of *k, l,* or *m,* can be reset within the range of the loop, either by arithmetic assignment statements or by READ statements;
5. None of *k, l, m* can be an arithmetic expression more complicated than just a constant or a single variable name.

Cautions:

1. If *l* is less than *k,* most FORTRAN compilers produce programs which traverse the loop once, nonetheless;
2. After a DO loop has run to normal completion, the value of the loop variable is undefined in machine memory.

On the whole, we pay rather heavily for the convenience of using the DO statement. We recall that loops can be organized by means of IF statements also. For such loops, none of the five restrictions applies, caution number two is not necessary, and caution number one is under the programmer's full control, to do as he pleases. Thus, the possibility of using an IF-coded loop, rather than a DO-coded loop, should never be forgotten. On the other hand, in many cases the DO statement achieves what we want, and leads to less complicated (and hence easier to debug) coding. In such cases, the DO statement should certainly be used.

Section C: Nested DO Loops.

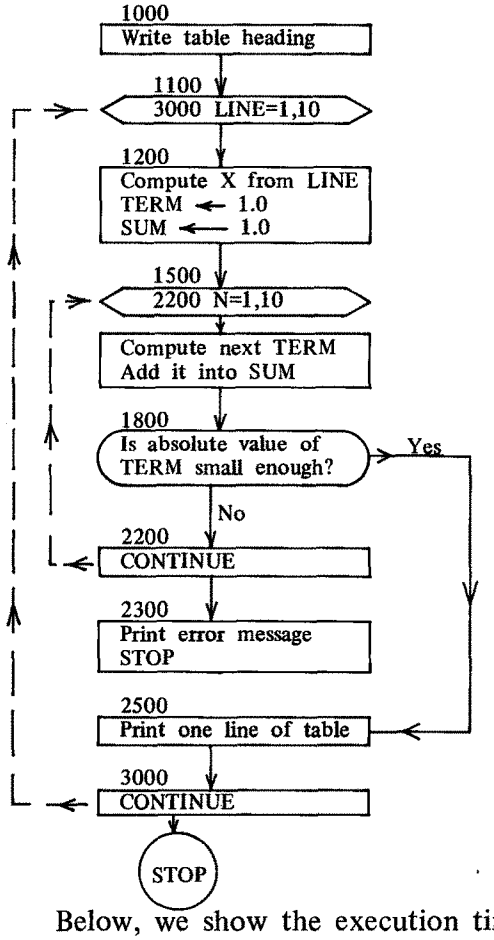
Let us now produce a small table of the function e^{-X}, rather than just a single value of the function. The table is to contain the function for x-values of 0.05, 0.10, 0.15, 0.20, ..., 0.50. This means that there will be ten lines in the table, the first line for x=0.05, the second line for x=0.10, ..., the tenth line for x=0.50. We shall organize this as a DO loop on the line number LINE, "around" the earlier program. The relation between the value of x and the number LINE of the line is simple: we get x by taking the line number, and multiplying by 0.05.

Looking at the flow diagram on page 87, we see that we can retain most of it for our new program. Alterations are required only at the beginning (the first box) and at the end.

In printing a table of numbers, it is best to confine the body of the table purely to numbers; the messages, which tell the reader what sort of table it is, should appear at the beginning, as a table heading. This table heading should be written out before we start the DO loop with LINE as the loop variable.

The new thing in the flow diagram on the next page is that we have a DO loop inside another DO loop. This is called "nested loops", and it is permitted in FORTRAN provided that the range of the inner loop is entirely contained within the range of the outer loop.

Whatever jumps occur, must either *remain within* the range of some DO loop, or must *remain outside* the range of some DO loop, or must *transfer out* of the range of a DO loop. They must never be into the range of some DO loop, from outside that loop. If there are several DO loops, nested or not, within one program, these conditions must be satisfied with respect to every DO loop in the program. For example, looking at the program below, consider the jump from statement 1800 to statement 2500. This is a jump out of the range of the inner DO loop; it is also a jump remaining within the range of the outer DO loop. On both grounds, it is a permissible jump.




```
C       TABLE OF EXP(-X)
        NPRNT=3
C       TABLE HEADING
 1000   WRITE (NPRNT,1010)
 1010   FORMAT(5X,1HX,9X,7HEXP(-X)/)
C       OUTER LOOP STARTS
 1100   DO 3000 LINE=1,10
 1200   EL=LINE
        X=0.05*EL
        TERM=1.0
        SUM=1.0
C       INNER LOOP STARTS
 1500   DO 2200 N=1,10
        EN=N
        TERM=-TERM*X/EN
        SUM=SUM+TERM
 1800   IF(TERM**2-1.E-12) 2500,2200,2200
 2200   CONTINUE
C       FAILURE OF ITERATION ...
 2300   WRITE (NPRNT,2310)
 2310   FORMAT(18H ITERATION FAILED  )
        STOP
C       ITERATION O.K. PRINT ONE LINE
 2500   WRITE (NPRNT,2510) X,SUM
 2510   FORMAT (F8.2,E16.7/)
 3000   CONTINUE
C       TABLE FINISHED.  GOOD-BYE.
        STOP
        END
```

Below, we show the execution time printed output from this program:

```
   X        EXP(-X)
0.05    0.9512294E  00
0.10    0.9048373E  00
0.15    0.8607078E  00
0.20    0.8187307E  00
0.25    0.7788007E  00
0.30    0.7408181E  00
0.35    0.7046879E  00
0.40    0.6703199E  00
0.45    0.6376280E  00
0.50    0.6065305E  00
```

Section D: The PAUSE Statement.

We have already introduced, and used, the STOP statement which terminates execution of the object program.

Sometimes it is desirable to interrupt the computation to permit the machine operator to take some action before the machine proceeds with the computation. For example, the operator may have to insert one of two different prepared sets of data cards, which set depends on the results so far; or the operator may be asked to mount or dismount a magnetic tape. For these purposes, a STOP statement is unsuitable, since it stops computation altogether.

The FORTRAN language contains a PAUSE statement for this purpose. Its effect is to stop program execution temporarily, until such time as the machine operator orders the machine to resume execution of the program. At this moment, the FORTRAN statement immediately following the PAUSE statement is obeyed next.

It is often necessary to give information to the machine operator, to tell him which of several courses of action he should take (for example, which of several prepared sets of data cards to insert into the card reader at this moment). The PAUSE statement permits a coded message to the operator, in the form of a string of from one to four digits, with the digits "8" and "9" forbidden. (The permitted digits, namely 0, 1, 2, 3, 4, 5, 6, and 7, can be thought of as the digits of some number expressed to base 8, rather than to base 10. However, this is not important at all here: the string of digits is simply a message to the operator; its numerical value is of no consequence). Within the program, we transfer control to one statement, say PAUSE 1111, if the operator is supposed to follow the first set of instructions, and to another statement, say PAUSE 2222, if the operator is supposed to follow the second set of instructions; and so on. The actual instructions (and data cards, if required) must be given to the operator beforehand, before the run is started.

Once a PAUSE statement has been reached, the machine waits for the operator to order it to start up again, by pressing the "start" button. Most large computer installations discourage the use of the PAUSE statement, because they claim that it wastes machine time: the machine stands idle while the operator does something or other. Even if the operator acts very rapidly, a large number of computations could have been carried out in the idle machine time.

In our view, this claim is debatable. The practice, now generally followed, of asking the programmer to prepare everything for the run in advance, and to leave no choice of action during program execution, frequently wastes considerable machine time. If everything has to be decided in advance, and if the programmer is forbidden to be at the machine while his program is being run, then it is possible to keep the machine busy doing nonsense on a program which has already effectively failed — only nobody knows it! This is particularly true of really large, complicated scientific and engineering calculations, somewhat less so in commercial data processing.

In the author's experience, it is impossible to do any effective large-scale computing in a "closed-shop" installation. If the user is not permitted near the machine, then for calculations of this sort the machine might as well not be there at all, for all the good it does.

With "multi-programming" coming to the fore, the PAUSE statement is likely to come into its own again. When a PAUSE statement is encountered in a multi-programming situation, the machine does not stop altogether. Rather, it stops work on your program, and starts to work on someone else's program. It resumes work on your program after the operator signifies that your program is once more ready for action. As far as you are concerned, the PAUSE statement does what it is claimed to do. But the machine is not in fact prevented from doing useful computations while waiting for the operator to act.

Many student compilers are written in such a way as to ensure rapid throughput of student jobs. Such compilers do not permit a student programmer to stop the machine, or to make it pause. With such compilers, the STOP statement causes the machine to progress to the next student job; and the PAUSE statement is equivalent, in its action, to the statement CONTINUE: The machine does not pause, and nothing is displayed to the operator.

Basic FORTRAN IV also permits a "message" on a STOP statement, of four digits, like in PAUSE.

Some installations prefer the statement "CALL EXIT" to "STOP", since STOP causes the machine to stop dead and just sit there waiting for the operator to do something. Find out for your installation!

Summary and Formal Definitions for Chapter VIII.

1. *Computed GO TO:*

   GO TO $(n_1,n_2,...,n_k),N$

   where n_1, n_2,...,n_k is a list of k statement numbers of executable statements in the program, and N is the name of an integer variable (not an array name). In execution, control is transferred to statement number n_1 is the value of N is 1; to statement number n_2 is the value of N is 2; ...; to statement number n_k if the value of N is k. The value of N must be one of 1, 2, 3, 4, ..., k.

2. *The CONTINUE statement:*

   This is a "no-action" statement; when supplied with a statement number, however, it can serve as the last statement in the range of a DO loop. This practice is recommended.

3. *The DO statement:*

   This takes one of two forms: DO *n J=k,l,m* or DO *n J=k,l*

   where *n* is the statement number of an executable statement farther on in the program, *J* is the name of an integer variable, and *k, l,* and *m* are each either an unsigned integer constant or an integer variable name. *k* is the initial value of *J, l* is the termination test value, *m* is the increment. In the second form of the DO statement, *m* is taken to be 1. None of *k, l,* or *m* may be zero, or negative. The *range* of the loop extends from the first executable statement following the DO statement, to statement *n* in the program. Statement *n* must be an executable statement, other than GO TO, IF, STOP, PAUSE, or RETURN. It may be (and we recommend) CONTINUE. Execution of the DO loop commences with *J=k.* Each time statement *n* is reached, *J* is increased by *m,* and this process continues until the new value of *J* exceeds *l,* or until a transfer occurs out of the range of the loop.

   Restrictions:

   1) *k, l,* and *m* must be positive (not zero or negative);
   2) *l* must be greater than or equal to *k*;
   3) It is forbidden to jump into the range of a DO loop from outside that range;
   4) Neither *J,* nor any of *k, l,* or *m,* can be reset within the range of the loop, either by arithmetic assignment or by READ statements;
   5) None of *k, l,* or *m* can be an arithmetic expression more complicated than just an unsigned integer constant or a single integer variable name.

   Cautions:

   1) If *l* is less than *k,* it is likely that the loop will be traversed once, anyway;
   2) After a DO loop has run to "normal completion" (no premature jumping out of the range of the loop), the value stored in memory location *J* (the value of the loop variable) is undefined and must not be used.

   Loops within loops (nested loops) are permitted, provided the range of any interior loop is a proper subset of the range of every outer loop around it.

4. *The PAUSE statement:*

   PAUSE or PAUSE *n*

   where *n* is an unsigned sequence of from one to four digits in the range 0 to 7, inclusive. Execution pauses until the machine operator tells the machine to recommence execution. At this point, execution resumes at the executable statement in the program immediately following the PAUSE statement.

DRILL EXERCISES FOR CHAPTER VIII.

1. Write a program segment for "flexible input". Each input card contains two numbers: a (real number) value in a field of width 10, followed by a field of width 5 containing an integer INDEX which is either 1, 2, 3, 4, or 5. The number is to be stored into location A1 if INDEX equals 1, A2 if INDEX equals 2, ...; we terminate on seeing a blank card. Use the IF and computed GO TO.

2. Correct the following statements for FORTRAN language errors, if necessary:
   a) DO 2413,I=1,K
   b) DO 3500 J=1,K+5
   c) DO 3600 J=2,L,B
   d) DO TILL 3600 J=2,KTHW7
   e) DO 3600 FOR J=1,99,3
   f) DO 3700 JJW=1,99,3,
   g) DO 3800 R=1,MAD
   h) DO 3900 J=I,J,K

3. In each of the following program segments, there is at least one error. Identify as many errors as you can:
   a)
```
      DO 1000 J=1,10,-1
      A=A+J**2
2000 J=J+1
1000 CONTINUE
      IF(I-5) 2000,2000,1000
```
   b)
```
      DO 1000 I=10,19
      A=I1+2*I
1000 DO 2000 J=2,9
2000 B=B+A**J
```
   c)
```
      DO 1000 I=1,J,I
9002 B=C+D**I
      CONTINUE
1000 IF(J-8) 3000,9002,9002
      K=I+J+7
3000 I=J-8
      GO TO 9002
```

4. Which of the following PAUSE statements are valid?
   a) PAUSE
   b) PAUSE +1234
   c) PAUSE 5678
   d) PAUSE 007
   e) PAUSE 1984
   f) PAUSE 70011
   g) PAUSE OPERATOR, PLEASE FIX MY PROGRAM

PROGRAMMING EXERCISES FOR CHAPTER VIII.

1. to 6. Rewrite programming exercises number 1, 2, 4, 5, 6, and 7 of Chapter VII, by replacing IF-coded loops with DO loops. The last two of these exercises are particularly important, since the calculation of primes requires nested DO loops, and the calculation of the power series requires jumping out of a DO loop before the loop variable has reached the termination test value.

7. The squares on a chessboard are designated by a row number, I, which may be any one of 1, 2, 3, ... 8; and by a column number, J, which may be any one of 1, 2, 3, ..., 8. A "bishop"moves diagonally across the board, in any direction. Write a program to read in I and J as input numbers, and count the number of permissible bishop's moves from the position (I,J); that is, all moves for which the final position is still on the chessboard. Recycle until a blank card is encountered.

8. Same as 7, but this time the chess piece is a knight rather than a bishop. The knight moves either (a) one place in a horizontal direction (right or left) and two places in the vertical direction, or (b) two places in the horizontal direction and one place in the vertical direction. Count the number of possible knight's moves from position (I,J). Recycle until a blank card is encountered.

9. Same as 7, again with a bishop, but this time read in two positions: (I,J) is the position of the bishop, and (K,L) is the position of another piece of the same colour. The rule in chess is that no two pieces of the same colour may occupy the same square, and a bishop must not move "across" one of the pieces of its own colour.

CHAPTER IX

BUILT-IN FUNCTIONS AND ARITHMETIC STATEMENT FUNCTIONS

Section A: Built-In Functions.

Every FORTRAN compiler provides certain "built-in" mathematical function evaluation programs. These contain the commands necessary to make the machine compute, starting from a number in some storage location, the desired function of that number. For example, the FORTRAN statement

```
Y=4.0+5.0*SQRT(X)
```

results in the following action: the machine fetches the real number contained in memory location X, and then transfers control to that region in memory which contains the commands required for evaluating the square root. These commands are obeyed, and the result is supplied back to the main program, ready to be used further. Control reverts to the main program. The number so obtained is now multiplied by 5.0, the number 4.0 is added to it, and the final result is stored into memory location Y.

There is no strictly *logical* necessity for this. In principle, every programmer could write his own set of orders to evaluate a square root (indeed, we have done just that already, several times over), and this set of orders could be inserted in every program which requires a square root, at every point where a square root is required. However, this would be incredibly inefficient, and would put an intolerable burden on the ordinary programmer. Hence every efficient master program, such as FORTRAN, must supply standard routines for evaluating the most important mathematical functions, and for doing certain other, very common, operations.

For each function, we must know the type (real or integer) of the argument of the function, and the type of the function value. The type of the function value is related to the name of the function in the usual FORTRAN fashion. Thus the name SQRT stands for the built-in square root function. Since the first letter is "S", the function value is a real number. Furthermore, the function SQRT expects a real number argument, e.g., SQRT(A) is all right, but SQRT(K) is grammatically incorrect FORTRAN.

Not only must the argument of the function be of correct *type* for that function (see the table a bit later), but many functions can not be evaluated at all for certain *values* of the argument. For example, the square root function can not be evaluated if the argument is negative (the square root of a negative number is not any real number). Different FORTRAN compilers do various things when an impossible argument is supplied for some function. With some systems, no warning is given at all, and some crazy function value is returned to the main program. Fortunately, such abominations are rare nowadays. Most systems do diagnose that an error has occurred, and provide execution time error messages to the programmer, of various degrees of obscurity.

Not all built-in functions are handled the same way in the machine. The more

complicated functions, such as the square root, are "external functions", for which the machine orders are kept in a separate region of memory, control being transferred to that region whenever that function is needed. But some other functions, for example the absolute value function, are "intrinsic functions". (FORTRAN jargon varies. Some people use "in-line" instead of "intrinsic", "out-of-line" instead of "external". Still other jargon terms exist). The absolute value of a number in the machine is obtained from that number in one single machine command, simply by forcing the sign to be "plus", no matter what it was before. This one machine command appears in the object program in the appropriate place; there is no transfer of control to some special location in memory. This distinction is of very little importance to the programmer.

We now present a table of built-in functions for USA Standard Basic FORTRAN IV. All but two of the functions in that table are functions of one argument; but the functions SIGN and ISIGN require *two* arguments, separated by a comma. The meanings of these two functions are:

SIGN(A,B) means the absolute value of A prefaced by the sign of B. For example, SIGN(3.5,2.1) equals 3.5, SIGN(-3.5,2.1) equals 3.5, SIGN(3.5,-2.1) equals -3.5, and SIGN(-3.5,-2.1) equals -3.5.

ISIGN(K,L) differs from the SIGN function only through the fact that both arguments, as well as the function value, are integers rather than real numbers.

TABLE OF BUILT-IN FUNCTIONS IN BASIC FORTRAN IV

Meaning of Function	*Name*	*Type of Argument*	*Type of Function Value*	*Intrinsic (I) or External (E)*
Absolute Value of Argument	ABS	real	real	I
	IABS	integer	integer	I
Natural Logarithm (Log to Base e)	ALOG	real	real	E
Exponential Function	EXP	real	real	E
Trigonometric Sine (Argument in Radians)	SIN	real	real	E
Trigonometric Cosine (Argument in Radians)	COS	real	real	E
Square Root	SQRT	real	real	E
Hyperbolic Tangent: tanh(x)	TANH	real	real	E
Inverse trigonometric tangent	ATAN	real	real	E
Conversion: Integer to Real	FLOAT	integer	real	I
Conversion: Real to Integer	IFIX	real	integer	I
Transfer of Sign (see above)	SIGN	real,real	real	I
	ISIGN	int.,int.	integer	I

The permissible range of the argument depends upon the function. For example, negative arguments are forbidden for the logarithm ALOG and for the square root SQRT. A zero argument is forbidden for the logarithm. The argument of the exponential function may be positive, negative, or zero; but if it is positive, it must not be so large that e^x exceeds the maximum permitted real number; this limits x to less than a few hundred.

The function FLOAT(K) has the effect of converting the integer K to real number form, and can be used as an alternative to the "implied conversion" statement GAY=K. Similarly, the function IFIX(A) has the effect of converting the real number A to an integer by the process of lopping off everything after the decimal point.

Functions can be used as arguments of other function. For example, SQRT(K), is grammatically incorrect since the SQRT function needs a real number argument. But SQRT(FLOAT(K)) is grammatically correct, and gives the square root of the number K converted to real number form in the machine.

The argument of the function need not be just a constant or a real variable name. On the contrary, it can be any arithmetic expression whatsoever, so long as the arithmetic expression is of the correct type (integer or real). For example,

```
Y=SQRT(B**2-4.0*A*C)
```

is a perfectly valid FORTRAN statement, which has the effect of setting Y equal to the square root of b^2-4ac.

Although the list of functions in the table is complete for Basic FORTRAN IV, most FORTRAN compilers provide quite a few additional built-in functions for the convenience of the programmer. Furthermore, those FORTRAN compilers which permit the use of double-precision real variables and/or of complex variables, also provide built-in functions for arguments of these other types. The built-in functions provided by your particular FORTRAN compiler are listed in Chapter XVII.

The built-in function SQRT(X) is not equivalent to X**0.5 in what it does. X**0.5 has the effect of taking the logarithm of X, multiplying that by 0.5, and then taking the antilogarithm to get the answer. SQRT(X) uses the iterative technique of Chapter III instead; this is faster and more accurate.

Another example in which use of built-in functions improves the efficiency of the program is in testing whether the absolute value of some quantity is small enough. So far, we have tested the square:

```
IF(RATIO**2-TOLER**2) 2500,2000,2000
```

transfers control to statement 2500 if and only if the absolute value of RATIO is smaller than TOLER. The statement

```
IF(ABS(RATIO)-TOLER) 2500,2000,2000
```

does the same thing, and does it faster, since forcing the sign of a number to "plus" (which is all that the function ABS does) is much faster in the machine than a multiplication.

Section B: Arithmetic Statement Functions.

Some functions which are not in the table of built-in functions can be expressed simply nonetheless. For example, the trigonometric tangent function is given by

$$\tan(x) = \sin(x)/\cos(x)$$

and can thus be expressed in terms of the built-in functions SIN and COS.(*)

(*) In many FORTRAN systems, TAN is itself a built-in function. But this does not change the principles involved here.

A programmer who needs a lot of values of the tangent, say tan(x), tan(u), tan(v), ..., may find it terribly tedious, and productive of coding errors, to have to write SIN(X)/COS(X) every time he wants TAN(X), to write SIN(U)/COS(U) every time he wants TAN(U), to write SIN(V)/COS(V) every time he wants TAN(V), and so on.

FORTRAN provides a special facility, called the *arithmetic statement function,* which permits the programmer to define, and then use, his own short-hand notation within his program. For example, the programmer may put, before the first executable statement of his program, the *arithmetic statement function definition:*

```
TAN(A)=SIN(A)/COS(A)
```

The identifier name A in this definition is called a *dummy argument;* its only real purpose is to tell the compiler that the *actual argument* (which can be any arithmetic expression whatever) is of real number (rather than integer) type.

For example, we may need the tangent of X+3.0*Y somewhere farther on in the program; we then merely write TAN(X+3.0*Y) in the appropriate place: the arithmetic expression X+3.0*Y is of real number type, and is therefore a valid actual argument of the function TAN. The effect is precisely as if we had written it out "in longhand" as: SIN(X+3.0*Y)/COS(X+3.0*Y).

The actual argument may be quite complicated, since it can be any arithmetic expression of the right type. For example, the statement

```
Y=U+V*TAN(X+3.0*TAN(Z))
```

is grammatically correct FORTRAN: the arithmetic expression "X+3.0*TAN(Z)" is of real number type, and that is all that is required. Without the use of the arithmetic statement function TAN, the statement would have had to be written out "in longhand" as:

```
Y=U+V*SIN(X+3.0*SIN(Z)/COS(Z))/COS(X+3.0*SIN(Z)/COS(Z))
```

The simplification resulting from the use of the shorthand notation TAN is obvious.

At first sight, it may seem that arithmetic statement functions are all that is ever needed. We can use such functions to define more complicated functions, and so on without limit. The earliest version of FORTRAN, called FORTRAN I, provided no other facility to the programmer to define his own functions.

However, the restriction that the definition of the function must be in the form of a single arithmetic statement, has turned out to be excessively severe! For example, the process used in the cube root program (programming exercise 5 of Chapter III, on page 34) involves an iterative loop requiring an IF statement for its termination test. Such a procedure can not be expressed in a single arithmetic statement, nor in terms of a sequence of arithmetic statement functions (arithmetic statement functions do not allow an IF statement within the procedure). This trouble is unfortunately very common.

To get around this, the next version of FORTRAN, called FORTRAN II, made it possible for the programmer to write his own external function subprograms, and all subsequent versions of FORTRAN, including FORTRAN IV, have retained this feature. FUNCTION subprograms are discussed in Chapter XII. From the practical point of view, everything that can be done with arithmetic statement functions can be done as well, or better, with external FUNCTION subprograms.

Summary and Formal Definitions for Chapter IX.

1. Built-in Functions:

If *W* is any well-formed arithmetic expression, and if *FNAME* is one of the standard names of FORTRAN built-in functions given on page 95, then *FNAME*(*W*) (or, for functions of two arguments, *FNAME*(*W,V*)) is also a well-formed arithmetic expression, whose value is that particular function of the value of *W* (of the values of *W* and *V*). Function evaluation takes precedence over all arithmetic operations in the rank order on page 18. The type (real or integer) of the function value is related to the function name in the usual way, through the first letter of the name. Furthermore, each function expects an argument (or two arguments) of definite type (see page 95).

Built-in functions may be "intrinsic" or "external". In the latter case, evaluation of the function involves transfer of control to a special region of memory, where the commands for evaluation of the function are stored.

2. *Arithmetic Statement Functions:*

   A) The *definition of an arithmetic statement function* must appear before the first executable statement of the program, and after the specification statements, if any (see Chapters X and XII for the meaning of "specification statements"). The defining statement has the form:

   NAME(A,B,...,E) = W

   where *NAME* is the function name and *A, B, ..., E* are *dummy argument names.* All of these must be grammatically correct identifier names, related to the variable type in the usual fashion. There must be at least one dummy argument. All dummy argument names must be different from each other, and they must be simple variable names, not array names. *W* is a well-formed arithmetic expression not containing array names or the names of not-yet-defined arithmetic statement functions.

   B) An *actual argument AP* is any arithmetic expression of the same type (integer or real) as the corresponding dummy argument name *A.*

   C) After an arithmetic statement function with a certain *NAME* has been defined, then *NAME(AP,BP,...,EP)* appearing anywhere later in the same program is a valid arithmetic expression, provided that *AP, BP,* etc. are valid actual arguments corresponding to *A, B,* etc., respectively. The value of this arithmetic expression is obtained by substituting, in the definition of the function *NAME(A,B,...,E)*, the value of *AP* for *A,* the value of *BP* for *B,* etc.

DRILL EXERCISES FOR CHAPTER IX.

1. Write arithmetic expressions, using built-in functions, for the following mathematical quantities (k and m are integers, other things are real numbers):
   a) a sin u - b
   b) a sin(u - b)
   c) $\sin\sqrt{\log_e(a^2 - b^3)}$
   d) $\cos(e^u+\sqrt{w})$
   e) tanh(arctan(u)+k)
   f) sin((x - m)/(x - 2m))

2. Write arithmetic statement function definitions for:
   a) The cotangent of an argument;
   b) The hyperbolic sine of an argument;
   c) The hyperbolic cosine of an argument.

3. The following program segments, each of which uses built-in functions or arithmetic statement functions, or both, all contain errors. See how many errors you can detect.

   a)
```
I=1
KSQU(I)=I**2
MIKE=KSQU(23+4*I)*KSQU(23-4*I)
```

   b)
```
EXPSQ(X)=EXP(-X**2)
X=10
Z=EXPSQ(X)*SIN(2)
Y=EXPSQ*Z
```

   c)
```
FUN(X)=EXP(ATAN(SQRT((X+1./X)**2)/2.)
U=2.5
Z=FUN(U)
ZZ=FUN(Z)
ZZZ=FUN(6.5)
ZZ=FUN(FUN(FUN(ZZ)))
```

CHAPTER X

ARRAYS

Section A: Vector Arrays.

We are often concerned with lists of similar items, for example, marks obtained by a student in different subjects. If a student takes physics, chemistry, mathematics, and biology, his four marks could be stored in four differently named memory locations, e.g., MARKP, MARKC, MARKM, and MARKB.

However, it is frequently more convenient to refer to the four subjects by numbers, say subject "1" is physics, subject "2" is chemistry, subject "3" is mathematics, and subject "4" is biology. We specify all four marks by the common identifier name MARK; if we want a particular mark, say the mark on the second subject (chemistry), then we enclose the subject number in parentheses: MARK(2). In the machine, the four numbers MARK(1), MARK(2), MARK(3), and MARK(4) appear next to each other in memory. The first memory location contains MARK(1), the second contains MARK(2), and so on. The entire set of four memory locations is called an *array*. More specifically, it is a *linear array* or *vector array*, of *dimension* equal to four.

The name MARK refers to the entire array. Any one storage location within this array contains one *array element.* Different array elements are distinguished by their *subscript.* MARK(2) has subscript equal to 2; it is the second array element in the array MARK. MARK(3) has subscript equal to 3; it is the third array element in the array MARK.

Before arrays and array elements can be used in a FORTRAN program, it is necessary to specify to the compiler that this particular identifier name is meant to identify an array, rather than a single number; and to specify the dimension of this array, so that the compiler known how much memory space to set aside for this array. These specifications are given by means of a *specification statement,* called a DIMENSION statement. Specification statements must come before all executable statements in the program, and before arithmetic function definitions (if any). In order to specify the name MARK to be the array name of a vector array of dimension four, we write:

```
DIMENSION MARK(4)
```

Note that the number "4" in parentheses here specifies the *dimension* of the array MARK; this number must be given as an unsigned integer constant. By contrast, later on in the program, the same character string "MARK(4)" refers to the biology mark (the fourth mark). Thus one and the same string of characters "MARK(4)" means one thing inside a DIMENSION statement, and something quite different in later, executable statements of the same program.

The type of number (real or integer) stored in an array element of the array MARK is related to the first letter of the array name in the usual fashion, i.e., all these array

elements contain integers, since the first letter of "MARK" is "M".

Once an array has been specified by a DIMENSION statement, both the array name MARK, and the name of a particular array element such as MARK(2), can be used for various purposes in the program, and will be understood correctly by the compiler.

To specify a particular array element, such as MARK(2), we require a *subscript.* This subscript may be, but need not be, an unsigned integer constant such as "2". It may instead be an integer variable name, such as the name "K" in "MARK(K)". The current value of K (i.e., the number contained in the memory location K at this moment) then decides which array element we want. For example, if K equals three at a certain moment, then MARK(K) is the third element of the array, same as MARK(3). If the value of K is altered to four later, then thereafter MARK(K) is the fourth element of the array, same as MARK(4). This technique permits us to refer to different array elements with a single name "MARK(K)", merely by altering the value of the subscript variable K.

Besides integer constants and integer variable names, FORTRAN allows certain other arithmetic expressions of integer type to be used as subscripts. However, there are limitations on the complexity of these *subscript expressions.* That is, it is not possible to use the most general expression of integer type. These rules are given in small print below. Most student programs are unlikely to violate these rules.

The following forms, and only those forms, are *valid subscript expressions* in USA Standard FORTRAN IV (either Basic or Full FORTRAN). In the table below, *k* and *m* stand for any unsigned integer constants, such as "2" or "526". *L* stands for any grammatically correct integer variable name, such as "K" or "KLOT" or "MUMPS".

Form No:	*Appearance*	*Typical example*	*Notes and Cautions*
1	*k*	MARK(2)	No plus sign; MARK(+2) is illegal!
2	*L*	MARK(KLOT)	
3	*L+k*	MARK(KLOT+2)	The order matters; MARK(2+KLOT) is illegal!
4	*L - k*	MARK(KLOT - 2)	Note that MARK(5-KLOT) is illegal!
5	*k*L*	MARK(2*KLOT)	Note that MARK(KLOT*2) is illegal!
6	*k*L+m*	MARK(2*KLOT+1)	Note that MARK(1+2*KLOT) is illegal!
7	*k*L - m*	MARK(2*KLOT - 1)	

The purpose of these rather severe restrictions is to enable the FORTRAN compiler to produce a more efficient object program (taking less machine time in execution) than what would be possible if one allowed the most general integer arithmetic expressions for the subscripts.

In actual fact, a large number of FORTRAN compilers in existence do permit much more general subscript expressions, many even the most general expression of integer or real type (if real, it is converted to integer form, by truncation of the fractional part, before being used). Nevertheless, we advise you strongly to stick to expressions of the seven standard forms listed above. Not only does this ensure compatibility of your program with all existing FORTRAN compilers, but it generally results in more efficient object code. After all, if a more complicated kind of subscript is needed, this can be achieved quite easily without violating the rules above, as follows: Suppose that we need array element number "K*L+M" of the array MARK. Since "K*L+M" is not one of the seven standard forms, we write:

```
KLM=K*L+M
MURK=MARK(KLM)
```

In practice, therefore, the limitations on subscript expressions are not at all troublesome.

Whether the subscript is an integer constant, or an integer variable name, or one of the other permissible subscript expressions, *the value of the subscript must lie between one and the declared dimension of the array.* For example, with four as the dimension of the array MARK, the value of the subscript can only be one of 1, 2, 3, or 4.

If the actual value of the subscript turns out to be zero or negative by mistake, then most operating systems detect that an error has occurred, and inform the programmer by means of an execution time error message, before terminating execution of the program.

But if the value of the subscript turns out to be larger than it ought to be (say, as a result of a mistake in the program, KLOT in MARK(KLOT) turns out to equal 6), then this error in the program is detected only by special student versions of FORTRAN. Other versions rarely test for *subscript overflow* in execution time.

Subscript overflow is a very nasty "bug" indeed. To understand what happens, let us observe that MARK(3), for example, is found in the machine as follows: we look to which memory location the name MARK "points"; this location contains the array element MARK(1); we now count down 1, 2, 3, and there is the memory location for MARK(3).

Now suppose that, as a result of a programming error, we encounter the statement MARK(KLOT)=0, at a moment when KLOT equals 6 in the machine. We go through the same procedure: we locate MARK(1), and count down 1, 2, 3, 4, 5, 6, and there is the memory location which we are told to set equal to zero; so that is what we do. And, lo and behold, a perfectly innocent memory location, having nothing whatever to do with the array MARK, has been cleared to zero! This memory location may have contained information of vital importance for the correct working of the program.

In aggravated instances, whole big areas of memory may be filled with completely erroneous information (such as zero), and some of these memory areas may have been intended to hold machine orders, rather than stored numbers. At some later stage in program execution, we attempt to obey these machine orders, only to find that they have been replaced by meaningless junk. The machine promptly stops execution with an error message saying that an illegal command has been encountered in a certain memory location. But that information is of very limited use. What we need to know is not which location in memory has been destroyed, but rather which machine command did the foul deed. The absence of this information is what makes this vile bug so terribly elusive.

There is another reason for considering subscript overflow as one of the nastiest of all bugs: during the process of debugging the program (see Chapter XIII) we are likely to employ only a small part of the array storage which we have reserved for normal program running. For example, if the program is designed to handle vectors of dimension up to 1000, we are likely to use much smaller vectors for checking purposes, that is, we may use only the first three or four array elements of the 1000-element array. Subscript overflow may then not occur. But when we use the "debugged" program for actual production running, we do employ most of the thousand locations in the array, and perhaps, inadvertently, a few more than that – and, pronto, subscript overflow is bugging us.

Little can be done to prevent this. You must be on the lookout for it, at all times, both when writing a program and while debugging it. And you can not afford to ignore the possibility of subscript overflow in a "debugged" program. We advise making checkruns with student compilers (e.g., WATFOR) in all cases, to catch subscript overflow.

Let us now turn to *input and output* of vector arrays. This can be done in two ways (a third way will be given in Chapter XI). We can specify an array element as an item in the list of a READ or WRITE statement; or we can specify the entire array. For example, consider the statements

```
5 FORMAT (I10)
  WRITE (NPRNT,5) MARK(2)
```

The effect is to print out the current value of the second mark, in I10 format. An array element in such a list is treated just like any other single variable: it points to a particular memory location (the second location within the array MARK) and the number to be printed is fetched from this location. On the other hand, we may also specify the name of the whole array, like this:

```
5 FORMAT (4I10)
  WRITE (NPRNT,5) MARK
```

This has the effect of printing out *all elements of the array, in the same order in which they appear in machine memory.* Thus the above statements are equivalent to:

```
5 FORMAT (4I10)
  WRITE (NPRNT,5) MARK(1),MARK(2),MARK(3),MARK(4)
```

Exactly the same rules apply to READ statements.

Section B: Finding an Item in an Ordered List.

As an important example of the use of vector arrays, consider the following problem:

In a certain country, incomes are arranged, for taxation purposes, into "tax brackets". The first tax bracket extends from zero income up to, say, $1000.00; incomes in that tax bracket are not taxed at all. Incomes in the second tax bracket, say between $1000.00 and $1500.00, are taxed at a certain rate. Incomes between $1500.00 and $2500.00 fall in the third tax bracket; and so on. The last tax bracket is bracket number 21, containing all incomes from $25000.00 on up, indefinitely. We are asked to write a program which will, first of all, input data values for these tax brackets, and store them away suitably in the machine memory. Having done so, we are to read the value of the taxable income from a data card, and determine into which tax bracket this taxable income TINC belongs.

All right, let us proceed to develop the solution to this problem. First of all, we must decide how to store the dividing incomes for the various tax brackets. We shall store them within a vector array BRACK. The first few array elements of BRACK are: BRACK(1)=0.0, BRACK(2)=1000.00, BRACK(3)=1500.00, BRACK(4)=2500.00, and so on, till BRACK(21)=25000.00. It will turn out convenient, later on, to reserve yet another element within the array BRACK, namely BRACK(22); and to store into BRACK(22) a value which is so high that all actual taxable incomes are smaller than that. We shall use the number 1.E+12 for this purpose.

First of all, the array BRACK must be specified at the very beginning of the program by means of

```
DIMENSION BRACK(22)
```

Next, we must read in data cards with values of the array elements. We do so by means of the statements

```
1000 READ (NREAD,1010) BRACK
1010 FORMAT (4E10.0)
```

This has the effect of reading in a number of data cards, each of which contains four array elements, at most. Since there are 22 array elements altogether, we read in six data cards. The first five data cards contain four numbers each, the last data card contains only two numbers. The E-format, rather than the F-format, has been chosen in order to permit input of the last number, equal to 1.E+12; this number could not be placed into an F10.0 field (explain!). True to our general principles, we follow this input of data values with an immediate echo-check of the array BRACK:

```
      WRITE (NPRNT,1020) BRACK
1020 FORMAT (13H TAX BRACKETS  /(4F20.2))
```

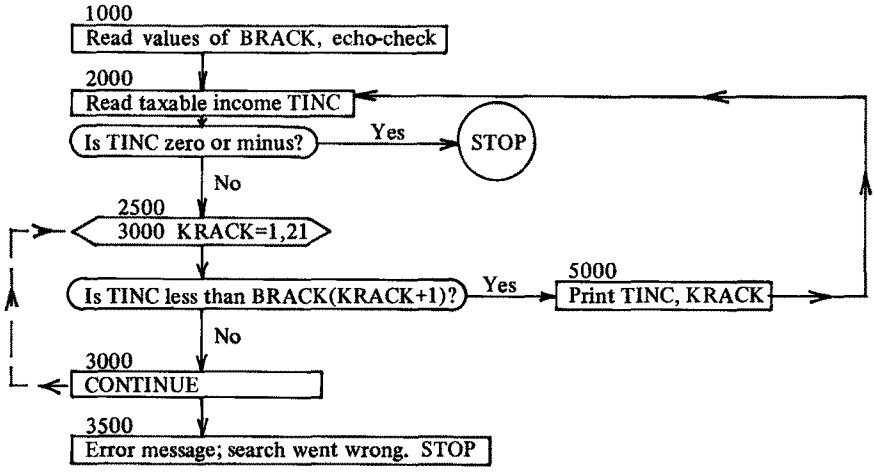
All right, this has solved the part of the problem dealing with input of the tax brackets.

Now let us proceed to the second part of the problem, namely, given a taxable income TINC read in from some data card, how do we determine the tax bracket into which this income belongs?

As an example, suppose that the income is $1635.80. This lies between BRACK(3) and BRACK(4) (between $1500.00 and $2500.00), and is therefore in tax bracket number 3. More generally, we shall need to search the list of tax brackets BRACK until we find a subscript value KRACK such that TINC is not less than BRACK(KRACK), and is less than BRACK(KRACK+1). If and when we find such a value of KRACK, then this value is the number of the tax bracket.

The advantage of storing a last, very high, value for BRACK(22) is that the last tax bracket, bracket number 21, is then no longer a special case. Rather, it fits within the general rule, since any reasonable income above $25000.00 lies between BRACK(21) and BRACK(22). Since "exceptional cases" are a terrible nuisance in programming, we gain quite a lot by using this little trick.

The simplest search procedure is a *straight sequential search.* We start by testing whether TINC is less than BRACK(2). If it is, KRACK equals 1, and we are finished. If not, we know that TINC is not less than BRACK(2), and we now compare TINC with BRACK(3). If it is less than BRACK(3), then KRACK equals 2, and the search is over. If not, we keep going. The flow diagram is shown below, and after it the FORTRAN coding.



```
C       SEARCHING SEQUENTIALLY THROUGH A LIST
        DIMENSION BRACK(22)
        NREAD=1
        NPRNT=3
C       READ THE TAX BRACKETS
 1000   READ (NREAD,1010) BRACK
 1010   FORMAT (4E10.0)
        WRITE (NPRNT,1020) BRACK
 1020   FORMAT (13H TAX BRACKETS  /(4F20.2))
C       READ DATA CARD WITH TAXABLE INCOME
 2000   READ (NREAD,1010) TINC
        IF(TINC) 2100,2100,2500
 2100   STOP
C       SET UP SEQUENTIAL SEARCH
 2500   DO 3000 KRACK=1,21
        IF(TINC-BRACK(KRACK+1)) 5000,3000,3000
 3000   CONTINUE
C       WE SHOULD NOT REACH THE STATEMENT BELOW AT ALL
 3500   WRITE (NPRNT,3600) TINC
 3600   FORMAT (36H SEARCH FAILED FOR TAXABLE INCOME =  ,E20.9//)
        STOP
C       WE REACH THIS POINT AFTER TINC HAS BEEN BRACKETED
C       BETWEEN BRACK(KRACK) AND BRACK(KRACK+1)
 5000   WRITE (NPRNT,5100) TINC,KRACK
 5100   FORMAT (9H INCOME =,F15.2,10X,13HTAX BRACKET =,I4)
        GO TO 2000
        END
```

Note that we should never get to statement 3500: since the last array element BRACK(22) has been set to an enormously high value, 1.E+12, TINC should at least be less than *that*! So we should jump out of the DO loop no later than with KRACK=21; if we fail to do so, and reach statement 3500 instead, something is very much amiss.

We now show some typical data cards, including one (next to last card) with an unreasonably high income.

Card No:	*First Field (Cols. 1-10)*	*Second Field (Cols. 11-20)*	*Third Field (Cols. 21-30)*	*Fourth Field (Cols. 31-40)*
1	0.0	1000.	1500.	2500.
2	3300.	3800.	4200.	4700.
3	5300.	6000.	6800.	7500.
4	8600.	9800.	10500.	12000.
5	14000.	16500.	18800.	21000.
6	25000.	1.E+12	blank	blank
7	1635.80	blank	blank	blank
8	250.00	blank	blank	blank
9	8700.	blank	blank	blank
10	24000.	blank	blank	blank
11	26000.	blank	blank	blank
12	1.E+13	blank	blank	blank
13	blank	blank	blank	blank

The first six cards are read by statement number 1000, and set BRACK. The others are read by statement number 2000, giving TINC. The printed output follows:

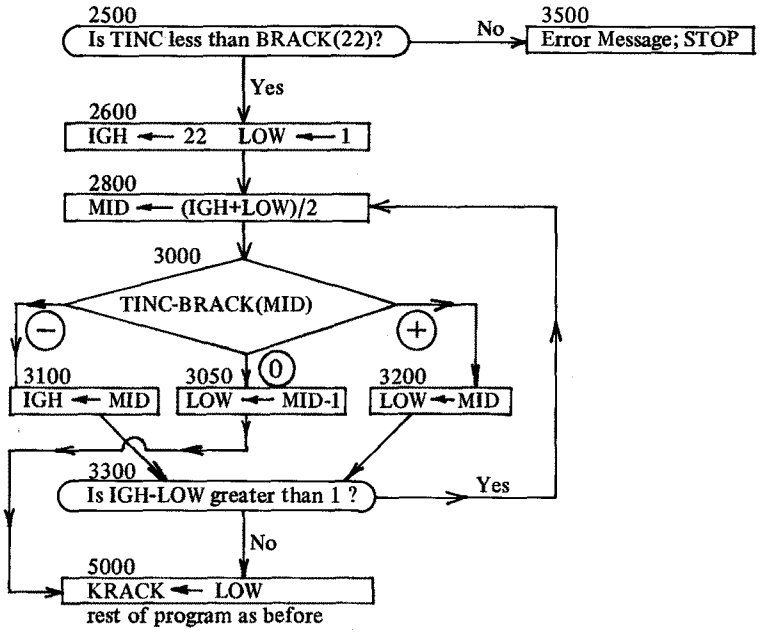
```
TAX BRACKETS
              0.00          1000.00          1500.00          2500.00
           3300.00          3800.00          4200.00          4700.00
           5300.00          6000.00          6800.00          7500.00
           8600.00          9800.00         10500.00         12000.00
          14000.00         16500.00         18800.00         21000.00
          25000.00  1000000000000.00
INCOME =     1635.80          TAX BRACKET =   3
INCOME =      250.00          TAX BRACKET =   1
INCOME =     8700.00          TAX BRACKET =  13
INCOME =    24000.00          TAX BRACKET =  20
INCOME =    26000.00          TAX BRACKET =  21
SEARCH FAILED FOR TAXABLE INCOME =       0.100000000E 14
```

Although this method of search works, it is inefficient. For example, when TINC equals 26000.00, then we must go through all the twenty-one separate tests in the DO loop, to find out that we are in the last tax bracket. On the average, we may have to perform some ten separate tests (in general, a straight sequential search through a list of N items requires N/2 steps, on the average). With only twenty-two items, this inefficiency is acceptable, in return for the simplicity of the program. But with long lists of thousands of items, which are by no means rare in commercial data processing, a better search strategy is required.

The *binary search* is a much more efficient strategy. We start by comparing TINC with the *middle item* in the list, i.e., with BRACK(11). Assume TINC is less than this. We then have TINC bracketed between BRACK(1) and BRACK(11). So we proceed to compare TINC with the middle item in that new range, i.e., with BRACK(6). Suppose that TINC is also less than BRACK(6), as it would be for TINC equal to 1635.80. Then we compare with the middle item in that new range, i.e., with BRACK(3). Since TINC exceeds BRACK(3), we then compare with the middle of *that* range, i.e., with BRACK(4). We find, for TINC equal to 1635.80, that TINC is less than BRACK(4). So now we have TINC bracketed tightly between BRACK(3) and BRACK(4). This took four comparisons, which for this particular income is slightly less efficient than the sequential search. But more careful consideration shows that we *never* need more than four comparisons, whereas we needed up to twenty-one comparisons for the sequential search.

Generally, if the number of items in an ordered list is greater than $2^{n-1}+1$ and less than or equal to 2^n+1, then the binary search takes at most n steps. As an example, let us take n=12. Then $2^{n-1}+1=2^{11}+1=2048+1=2049$, and $2^n+1=2^{12}+1=4096+1=4097$. Thus, if our list has more than 2049 items in it, but no more than 4097 items, then the binary search technique allows us to find any item in that list in no more than 12 steps. By comparison, a sequential search through a list of, say, 4000 items would require some 2000 steps on the average, and could require as many as 4000 steps!

We now present that part of the flow diagram, and of the FORTRAN code, which is changed from before, i.e., which replaces statements 2500 to 5000 inclusive of the earlier version. At any given point during the binary search, we know two subscripts IGH and LOW, such that BRACK(LOW) is less than or equal to TINC, and TINC is less than BRACK(IGH). If IGH-LOW equals one, then we are finished, and KRACK=LOW is the desired bracket number. If not, we test TINCagainst BRACK(MID), where we set MID=(IGH+LOW)/2. If TINC is less than BRACK(MID), we set IGH=MID. If TINC is greater than BRACK(MID), we set LOW=MID. Finally, if TINC is exactly equal to BRACK(MID), then there is no point in searching further: the desired bracket number is then equal to MID-1, by our definition (that is, TINC is then greater than BRACK(MID-1) and less than or equal to, in fact equal to, BRACK(MID)).




```
C       FIRST CHECK WHETHER TINC IS LESS THAN BRACK(22)
 2500 IF(TINC-BRACK(22)) 2600,3500,3500
C       YES, IT IS. SET UP BINARY SEARCH.
 2600 IGH=22
      LOW=1
C       LOOP STARTS HERE
 2800 MID=(IGH+LOW)/2
 3000 IF(TINC-BRACK(MID)) 3100,3050,3200
 3050 LOW=MID-1
C       THIS VALUE OF LOW WILL BE USED IN STATEMENT 5000
C       THIS IS THE CORRECT VALUE FOR TINC=BRACK(MID)
      GO TO 5000
C       THE BRANCH BELOW IS FOR TINC LESS THAN BRACK(MID)
 3100 IGH=MID
      GO TO 3300
C       THE BRANCH BELOW IS FOR TINC GREATER THAN BRACK(MID)
 3200 LOW=MID
C       THESE TWO BRANCHES REJOIN HERE
 3300 IF(IGH-LOW-1) 5000,5000,2800
C       ERROR BRANCH.  FAILURE MESSAGE, THEN STOP.
 3500 WRITE (NPRNT,3600) TINC
 3600 FORMAT (36H SEARCH FAILED FOR TAXABLE INCOME =  ,E20.9//)
      STOP
C       BINARY SEARCH SUCCESSFULLY COMPLETED.
 5000 KRACK=LOW
      WRITE (NPRNT,5100) TINC,KRACK
C       FROM THIS POINT ON, SAME AS ON PAGE 104
```

The earlier program is taken over unchanged down to statement 2500, and after statement 5000. The data cards are unchanged also, and so is the output.

If the elements of the list are real numbers, as here, it is rare to find exact equality for a comparison such as in statement 3000. Thus, we would jump to statement 3050 only in exceptional cases. But if the list elements are integers, exact equality is not so unusual, and the branch to 3050 can save quite a bit of machine time.

Note on Diagnostics:

A particularly frequent coding error is omission of the DIMENSION statement for an array. This gives rise to diagnostic messages from the FORTRAN compiler. But the nature of these messages may cause some surprise: all of them say something about *illegal statement functions* or *missing subprograms*!

The reason is this: When the FORTRAN compiler encounters the character string "MARK(I)", say, it first looks whether there is a DIMENSION statement for an array by the name of MARK. If not, the compiler looks whether an arithmetic statement function with name MARK has been defined earlier in this program, or whether perhaps the statement now being translated can be interpreted as an attempt to define an arithmetic statement function. A perfectly innocuous assignment statement such as

```
MARK(I)=MARK(I)+1
```

is interpreted as such an attempt (in the absence of a DIMENSION statement for MARK); as such, the attempt is invalid, since the function to be defined, MARK, appears on the right side of the equal sign as well as on the left side, i.e., the function is defined in terms of itself. This gives rise to a diagnostic message. Or else, the diagnostic message might refer to the fact (if it is a fact) that this statement comes later than the first executable statement of the program, something which is forbidden for definitions of arithmetic statement functions.

On the other hand, a statement such as K=MARK(I) could not possibly be an attempt to *define* an arithmetic statement function. For that, the function name MARK would have had to appear to the *left* of the equal sign. However, MARK could be the name of a built-in function, or of a programmer-supplied FUNCTION subprogram (see Chapter XII). The compiler assumes so, and bides its time. But finally, after all the FORTRAN statements have been translated into machine language, and it is time for program execution, at this point the compiler notices that there is something amiss: in fact, there is no built-in function by the name of MARK, nor is there a programmer-supplied FUNCTION subprogram by that name. So, the monitor system issues a diagnostic message to the effect that a subprogram by the name of MARK can not be found.

Section C: Matrix Arrays.

In addition to vector arrays, for which the array elements have single subscripts, FORTRAN also allows the use of "matrix arrays", where each array element requires two subscripts for its specification. For example, suppose that we have 50 students in a class, and each student takes four subjects. It is then natural to allocate the storage location MARK(45,3), say, to the mark of student number 45 on subject number 3. We can set out these marks in a table, each line of the table for one student.

Such a table is shown on the next page. It is called a "matrix array". Each entry in the table is a "matrix element". The total number of matrix elements in this table (not all of them are shown in the diagram) is 50*4=200. Thus, to store this set of numbers in the machine, we require 200 memory locations.

The DIMENSION statement of the FORTRAN language is used to inform the compiler that MARK is meant to be a matrix array, of total size 50-by-4:

```
DIMENSION MARK(50,4)
```

This tells the compiler to reserve 200 memory locations for the matrix array MARK.

SCHEMATIC PICTURE OF A MATRIX ARRAY

	Subject 1	*Subject 2*	*Subject 3*	*Subject 4*
Student 1	MARK(1,1)	MARK(1,2)	MARK(1,3)	MARK(1,4)
Student 2	MARK(2,1)	MARK(2,2)	MARK(2,3)	MARK(2,4)
Student 3	MARK(3,1)	MARK(3,2)	MARK(3,3)	MARK(3,4)
Student 4	MARK(4,1)	MARK(4,2)	MARK(4,3)	MARK(4,4)
Student 5	MARK(5,1)	MARK(5,2)	MARK(5,3)	MARK(5,4)
.....	.....	.....	.....	.....
Student 49	MARK(49,1)	MARK(49,2)	MARK(49,3)	MARK(49,4)
Student 50	MARK(50,1)	MARK(50,2)	MARK(50,3)	MARK(50,4)

The general array element is designated MARK(I,J) in the FORTRAN program. The DIMENSION statement at the beginning of the program gives the compiler enough information, not only to know how many storage locations (200) to set aside for this array, but also how to find every array element MARK(I,J) later on in the program. The first subscript, I, can take the values I=1, 2, 3, ..., 50; the second subscript, J, can take the values J=1, 2, 3, 4.

It is sometimes important to know just how FORTRAN arranges the matrix elements within this storage area of 200 locations. One might be tempted to think that they are stored in the same order in which we would read the above table, i.e., reading from left to right, line after line. This is *not* the method used for FORTRAN matrix storage.

Rather, *FORTRAN arranges storage column by column.* The entire first column of the matrix is stored first. Next comes the entire second column of the matrix; and so on. Below, we show schematically the sequence of storage locations for the array, together with the consecutive order numbers 1, 2, 3, ..., 200:

Order Number in Storage	*Matrix Element Stored There*
1	MARK(1,1)
2	MARK(2,1)
3	MARK(3,1)
..	.....
50	MARK(50,1)
51	MARK(1,2)
52	MARK(2,2)
53	MARK(3,2)
....	.....
100	MARK(50,2)
101	MARK(1,3)
102	MARK(2,3)
.....	.....
150	MARK(50,3)
151	MARK(1,4)
152	MARK(2,4)
.....	.....
200	MARK(50,4)

We can represent the relationship between the order number and the subscripts I and J of MARK(I,J) by a simple formula, namely

Order Number for storage of MARK(I,J) = I + 50*(J - 1)

where the "50" is taken from the first number appearing in the specification statement DIMENSION MARK(50,4). More generally, the order number is given by I+*n**(J-1) where *n* is the *first* integer constant appearing in the DIMENSION statement for that array. This order number is computed by the FORTRAN-compiled object program every time we refer to MARK(I,J) in the program.

The way in which matrices are stored is particularly important when we wish to read, or write out, an entire matrix array. For example, the statements

```
      WRITE (NPRNT,1200) MARK
1200  FORMAT (12I10)
```

have the effect of printing out all 200 array elements of the array MARK, twelve to a line, in the order in which the elements are stored in the machine. For vector arrays, the order of storage in the machine is the "natural" order, but for matrix arrays, the matrix elements appear in the wrong sequence. Furthermore, with so many lines of output, and so many elements to a line, it is easy to lose track. For these reasons, *this form of matrix array output is not recommended.*

Similar objections apply to the use of the array name in a READ list. For example,

```
      READ (NREAD,1300) MARK
1300  FORMAT (24I3)
```

has the effect of reading in all 200 marks, 24 marks to an input card. But the input has to be prepared in the peculiar order preferred by FORTRAN, and mistakes are only too likely. Better methods for organizing reading and writing of whole matrix arrays, or parts of such arrays, will be presented in Chapter XI.

None of these objections apply to reading or writing of some particular array element. For example, MARK(25,3) may be specified within a simple list in a READ or WRITE statement. This is treated like any other list element, i.e., it is the name of a storage location into which some number is read, or from which some number is fetched for output.

Matrix arrays take a great deal of room in storage, and should be used with care. It is often possible, in data processing, to avoid the use of matrix arrays by reorganization of the program. As an example, consider the following problem:

Example: There are 50 students, each taking the same four subjects. The marks are to be read in, and the class average computed for each of the four subjects. The grand total average is to be computed as well.

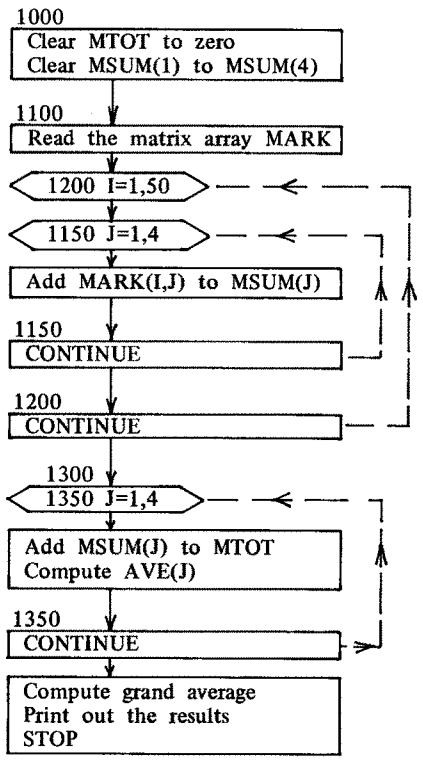
First Solution: We use a matrix array MARK, of dimension 50-by-4, to store the marks, and then we operate on this array. A vector array MSUM, of dimension 4, is to hold the sums of the marks in each of the four subjects. We use nested DO-loops to accumulate the mark-sums. The flow diagram and the code are given on the next page.

This solution will work (incidentally, can you explain why the alternative statement AVE(J)=MSUM(J)/50 placed just before statement 1350 in the program would not work at all?).

But even though it works, this solution has severe disadvantages.

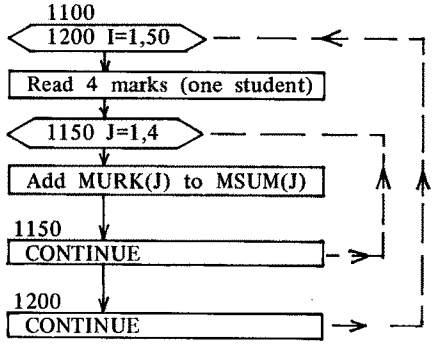
(i) We need matrix array storage for a 50-by-4 matrix; and

(ii) The marks have to be prepared on data cards in the peculiar sequence favoured by FORTRAN. Thus the marks of any one student do not appear together on one input card.




```
C       CLASS AVERAGES, MATRIX FORM
        DIMENSION MARK(50,4),MSUM(4),AVE(4)
        NREAD=1
        NPRNT=3
 1000   MTOT=0
        DO 1010 J=1,4
        MSUM(J)=0
 1010   CONTINUE
C       NOW START REAL WORK.  FIRST READ.
 1100   READ (NREAD,1120) MARK
 1120   FORMAT (8I5)
C       NOW ACCUMULATE MSUM VALUES
        DO 1200 I=1,50
        DO 1150 J=1,4
        MSUM(J)=MSUM(J)+MARK(I,J)
 1150   CONTINUE
 1200   CONTINUE
C       NOW PROCESS THE RESULTS
 1300   DO 1350 J=1,4
        MTOT=MTOT+MSUM(J)
        AVE(J)=FLOAT(MSUM(J))/50.0
 1350   CONTINUE
        GRAND=FLOAT(MTOT)/200.0
        WRITE (NPRNT,1400) AVE,GRAND
 1400   FORMAT (9H AVERAGES  ,4F8.1,F15.1)
        STOP
        END
```

Second Solution: There is actually no need to "remember" all the marks at once. It would be better to read in fifty cards, each card containing the marks of one student on his four subjects, and to process the information on each card immediately, so that it can be "forgotten" afterwards. With this second method, we shall need only a vector array MURK, of dimension four, instead of the matrix array MARK. The outer parts of the flow diagram, and of the code, are the same. But the section from statement 1100 to statement 1200, inclusive, is replaced by:




```
C       CLASS AVERAGES, VECTOR FORM
        DIMENSION MURK(4),MSUM(4),AVE(4)
        ......
 1100   DO 1200 I=1,50
        READ (NREAD,1120) MURK
 1120   FORMAT (4I5)
        DO 1150 J=1,4
        MSUM(J)=MSUM(J)+MURK(J)
 1150   CONTINUE
 1200   CONTINUE
        ......
```

The gain in storage locations, as well as in ease of preparation of input cards, is enormous.

We repeat: In all data processing, think of the operation as a whole, not merely the computer program as such. The operation as a whole includes the problem of preparing the input data cards in an easy and flexible fashion. Try to process the input data as they arrive, as far as possible, so as not to tie up valuable space in machine memory with material which need not be "remembered" at all, if one only thinks a bit about how best to organize the flow of the program.

Section D: The EQUIVALENCE Statement.

In more advanced computer programming, it is sometimes desirable to give different identifier names to one and the same memory location in the machine. It may be possible to save storage locations by using one location for two or more different purposes; or it may be possible to force the FORTRAN compiler to produce a faster-running object program, by referring to one and the same memory location in two different ways.

For example, let us suppose that a certain program needs three vector arrays, JACK of dimension ten, HARRY of dimension thirteen, and JOE of dimension fifteen. The program uses HARRY first; thereafter, only the first element of this array, HARRY(1), need be "remembered". We wish to refer to this particular storage location by the alternative name HAL, thereby saving machine time, since the machine takes longer to "fetch" an element of a vector array, than to fetch a number from a non-array location. The vector array JACK is used next in the program, and after it has been used, only the second array element, JACK(2), need be remembered. We shall call this one JACKY. Finally, the program uses the vector array JOE. We want to save as much storage as we can, without overwriting storage locations whose contents we must not "forget". Without doubling up, we would need a total of 10+13+15=38 storage locations for the three arrays. However, the following scheme shows that we can compress this to 18 locations by using more than one name for one and the same location.

Location	*Names*
1	HARRY(1),HAL
2	HARRY(2),JACK(1)
3	HARRY(3), JACK(2), JACKY
4	HARRY(4), JACK(3), JOE(1)
5	HARRY(5), JACK(4), JOE(2)
..	.....
11	HARRY(11), JACK(10), JOE(8)
12	HARRY(12), JOE(9)
13	HARRY(13), JOE(10)
14	JOE(11)
15	JOE(12)
....	.....
18	JOE(15)

You should check through this scheme to see that quantities which must be remembered later are not in fact overwritten; for example, since we must remember JACKY, the vector array JOE must not use that location in memory.

To achieve this doubling up of storage, we write the following FORTRAN specification statements:

```
DIMENSION JACK(10),HARRY(13),JOE(15)
EQUIVALENCE (HAL,HARRY(1)), (HARRY(3),JACK(2),JACKY), (JACK(3),JOE(1))
```

The first "equivalence group" (HAL,HARRY(1)) declares that HAL and HARRY(1) refer to the same storage location in the machine. The second equivalence group declares HARRY(3), JACK(2), and JACKY to be synonyms for one storage location. Since HARRY and JACK are both vector arrays, the entire arrays are positioned in machine memory in accordance with this declaration, i.e., not only are HARRY(3) and JACK(2) one storage location, but HARRY(2) is the same location as JACK(1), HARRY(4) is the same location as JACK(3), etc., etc. The third equivalence group (JACK(3),JOE(1)) positions the first array element JOE(1) of the vector array JOE into the same memory location as the array element JACK(3). All other array elements of JOE are therefore positioned accordingly, as well. The net result is the one shown above.

Note that the two FORTRAN statements A=HARRY(3) and A=JACK(2) do quite different things, in spite of the fact that a number is fetched initially from one and the same storage location. The statement A=HARRY(3) is translated into a fetch, followed immediately by storage of the fetched number into location A. But when the FORTRAN compiler encounters the statement A=JACK(2), it notes the fact that the name JACK starts with "J", and therefore JACK(2) is interpreted as an integer. The statement A=JACK(2) involves an implied conversion from integer to real number form. Thus the number finally stored into A is quite different from the number HARRY(3). "Equivalence" in the FORTRAN sense means the same location in storage, nothing more. It is in no sense a mathematical equivalence.

Summary and Formal Definitions for Chapter X.

1. An *array* is a sequence of contiguous storage locations in machine memory. Each storage location in the array is called an *array element.* An array is given an *array name* which is a standard identifier name, with the standard relation between the type (real or integer) of the numbers in the array and the first letter of the array name. An array element is identified by following the array name with one (for "vector arrays") or two (for "matrix arrays") *subscripts,* the subscripts being enclosed within parentheses.

2. A *vector array declarator* has the form *NAME(n)* where *NAME* is the array name, and *n* is an unsigned integer equal to the total number of storage locations in the array, and called the *dimension* of the vector array.

   A *matrix array declarator* has the form *NAME(n,m)* where *NAME* is the array name, and both *n* and *m* are unsigned integer constants. The total number of storage locations set aside for this array is the product *n*m.*

3. The DIMENSION statement is a specification statement (non-executable) for declaring array names and their dimensions. It has the form:

   DIMENSION *D,D',D'',...,D'''*

   where *D, D',* etc. are array declarators, separated by commas.

4. A *subscript expression* must be an arithmetic expression of integer type, and must have positive (not zero or negative) integer value. Furthermore, subscript expressions are of limited complexity. Letting *k* and *k'* be unsigned integer constants, and letting *J* be an integer variable name, the permitted subscript expressions are:

   *k J J+k J-k k*J k*J+k' k*J - k'*

   The precise *order* matters here: for example, *k+J is not permitted.*

5. Let *NAME* be the name of a vector array, of dimension *n,* and let *k* be a permissible subscript expression, whose actual value lies between 1 and *n,* inclusive. Then *NAME(k)* is the *k*'th storage location within the array *NAME.*

   Let *NAME* be the name of a matrix array, with first dimension *n* and second dimension *m.* Let *j* be a permissible subscript expression with actual value between 1 and *n.* Let *k* be a permissible subscript expression with actual value between 1 and *m.* Then *NAME(j,k)* is the *i*'th array element of the array *NAME,* where the *order number i* is given by: $i=j+n*(k-1)$.

6. An *equivalence group* has the form (*List*) where *List* is a simple list of variable names and/or array element identifiers. The list must have at least two elements. If a list element is an array element identifier (array name followed by subscript or subscripts in parentheses), then the subscript or subscripts must be unsigned integer constants; that is, in this context the only permitted subscript expression is *k* in (4) above.

   Within an equivalence group, but only within an equivalence group, a matrix element *A(k,k')* belonging to a matrix array *A* with dimensions *n* and *m,* may be referred to by its order number *i* directly. That is, we may use *A(i)* where $i=k+n*(k'-1)$ is the order number. This form of reference is likely to lead to coding errors, and is therefore not recommended.

7. The EQUIVALENCE statement has the form

   EQUIVALENCE *G,G',G'',...,G'''*

   where *G, G',* and so on are equivalence groups. The named variables and/or array elements within any one equivalence group are thereby declared to be synonyms for one memory location in the machine. If an array element appears within an equivalence group, then all the other array elements of the same array are positioned in machine memory accordingly, following their order numbers within the array. Equivalence implies only the same location in machine memory, not mathematical equality.

8. In a FORTRAN program, all DIMENSION statements must precede all EQUIVALENCE statements, all EQUIVALENCE statements must precede all arithmetic function definitions (if any); and all arithmetic function definitions must precede the first executable statement of the program.

DRILL EXERCISES FOR CHAPTER X.

1. Given the following short program segment:

```
      DIMENSION A(5),LOON(5),B(5)
      DO 1200 I=1,5
      A(I)=I**2-3*I
      LOON(I)=IABS(I-2)+1
      J=LOON(I)
      B(J)=I+J
1200 CONTINUE
```

   State the content of those vector array elements which have been set in this loop, and state which elements, if any, have not been set.

2. Write FORTRAN statements to do the following:
   a) The vector array G has 15 components, which are to be permuted cyclically: G(2) is to contain the value formerly stored in G(1), G(3) is to contain the value formerly stored in G(2), ..., G(15) is to contain the value formerly stored in G(14), and G(1) is to contain the value formerly stored in G(15).
   b) In a list of N values, B(1), B(2), ..., B(N), with N odd, interchange the value of the last item with the value in the middle of the list.
   c) In a list of twenty values, C(1) is to be exchanged with C(2), C(3) with C(4), and so on, until C(19) is exchanged with C(20).

3. Declare the array KOOKY to be 8-by-8, and write commands to make KOOKY(I,J) equal to I+J if I is odd and J is even, equal to I-J is I is even and J is odd, and zero otherwise.

4. Let A and B be vector arrays, with thirteen components each. Let Q be a simple (non-array) variable name. Write FORTRAN program segments for the following operations:
   a) Multiplication of a vector by a scalar: put Q times A(I) into B(I).
   b) Add the two vectors, element by element, placing the resulting sum into A.
   c) Subtract vector B from vector A, component by component, placing the resulting vector array into the B vector locations.
   d) Find the sum A(1)*B(1)+A(2)*B(2)+.....+ A(13)*B(13) and place it into location Q(this sum is called the "scalar product" of the vectors A and B).
   e) The "length" of a vector is defined to be the square root of its scalar product with itself. Find the length of the vector A, and store it into Q.

5. Repeated product: Store in WHY the value of the product of the first N vector components A(I) of a vector array dimension one hundred (assume that N is less than 100).

6. Reverse the order of the elements B(J) for J=1, 2, ..., M. That is, interchange B(1) with B(M), interchange B(2) with B(M-1), and so on.

7. A checker board (not a chess board) is described by an eight-by-eight matrix array MORDE(I,J) in the following way: The value of MORDE(I,J) is +1 if a "white piece" is in position (I,J) on the board (in row I and column J); MORDE(I,J) is +2 if a "white king" is in that position; -1 if a black piece; -2 if a black king; and 0 if nothing is in that position.
   a) Write a program to count the white kings, the count to be placed into KOUNT.
   b) Same as (a), but for black pieces.
   c) Count the number of white kings on the main diagonal of the board (I=J).
   d) Count the number of black kings below the main diagonal (J less than I).
   e) Count the number of white pieces which are not on the main diagonal.
   f) (For checkers enthusiasts only) Count the number of black pieces which are required to "jump" over a white piece,or king, in this position of the board; assume it is black's move.

8. We want to read the present position of the checker board into the array MORDE by means of the statements:

```
      DIMENSION MORDE(8,8)
      READ (NREAD,1200) MORDE
1200 FORMAT (8I3)
```

   Explain in detail how the data cards must be prepared, particularly the sequence in which the various positions on the board are to be coded into each data card.

9. Given a checker board position stored in the array MORDE, what would be wrong with printing it out by means of the statements of problem 8, with the middle statement replaced by
WRITE (NPRNT,1200) MORDE

10. Assume that the checker board array MORDE is already stored in machine memory, and we have a data card prepared as follows: There are nine fields on the card, each of width three; the first field contains a row number (between 1 and 8), and the remaining eight fields contain new information (+2, +1, 0, -1, or -2, as the case may be) concerning the white and black pieces and kings across that row. Write a program segment to read this card, and to store the new information for this row in the right place within the array MORDE.

11. (More general version of problem 10) If the first field contains a negative number (between -1 and -8), then the remainder of the card refers to one *column* of the checker board, not one row. The column (or row) number is the absolute value of the number appearing in field one of the data card. Insert the column (or row, as the case may be) in the correct position within the array MORDE.

12. Use a vector array JOKER, of dimension eight, and some DO loop coding so as to produce a reasonable printout of the checker board position, i.e., row 1 on the top line, nicely spaced out, then three blank lines, then row 2, then another three blank lines, and so on down to row 8.

13. Write suitable DIMENSION and EQUIVALENCE statements for the specifications given below:
    a) Array B is 11-by-11 and the vector array VEC, with 11 components, coincides with the first column of the matrix array B.
    b) Same as (a), but VEC coincides with the last column of B.
    c) Explain why it is *impossible* to make VEC coincide, element by element, with the second *row* of B.
    d) A matrix array CARDS has four rows, each of length thirteen. The last three columns of the array are to have the names QUEEN, QUING, and ACES, respectively.

14. Find the errors in the following program:

```
1000 DIMENSION YA(3,5),YAHOO(35),NOHOW(14,0)
1100 DIMENSION VIC(-32),WAG(20000,3000000),WOG(3,2,4,8)
1120 I=2
1140 J=3
1160 K=8
1200 DIMENSION WOOLY(K)
1300 NREAD=1
1400 READ (NREAD,1500) YAHOO
1500 FORMAT (8I3)
1600 A=YAHOO(3*I+J)
1700 B=YAHOO(I,J)
1800 C=YAHOO(I-2)
1900 D=YAHOO(35-I)
2000 YA(I,J)=YAHOO(K)
2100 YA(I,K)=YAHOO(J)
2200 YA(K,J)=YAHOO(I)
2300 GO TO 1000
2400 STOP
     END
```

PROGRAMMING EXERCISES FOR CHAPTER X.

1. *Sorting:* Read a vector array A, of dimension ten, from a data card; echo check; then rearrange the elements in non-decreasing sequence, that is, so that A(1) is no larger than A(2), A(2) is no larger than A(3), and so on. Use the following methods in turn:
    a) "Ripple-Sort" (also. called "bubble-sort"): On the first run through, compare A(1) with A(2), and interchange them if A(1) is larger than A(2); then compare A(2) with A(3), and interchange them if A(2) is larger than A(3); and so on, till a comparison of A(9) with A(10). Keep track of whether any interchange proved necessary, and if so, which was the last interchange needed during this run. If no interchange was needed, the sorting is finished. If an interchange had to be made between A(LAST) and A(LAST+1), then we

repeat the process of comparison, starting again with A(1) versus A(2), but this time going only as far as comparing A(LAST-1) with A(LAST). (Can you explain why one need not go any farther?) Keep going until either no interchange was needed, or until LAST equals one. Then print out the sorted vector. (Hint: During debugging, it pays to print out the partially sorted vector at the end of each "pass" of this process).

b) "Ranking Sort": Reserve storage space for a second vector, say B. Start by putting A(1) into the B(1) location. Now compare A(2) with B(1); if A(2) is the larger one, it goes straight into B(2); otherwise, we move B(1) into the B(2) location, and insert A(2) into the B(1) location. In either case, B(1) and B(2) are in increasing order. Now compare A(3) first with B(2), then with B(1). Whenever a comparison "fails" (in the sense that the element of the array B is bigger than the element from A), the B element is moved one place up in the B array; as soon as a comparison "succeeds", the element from A is inserted into the vacant spot. For example, if A(3) is smaller than B(2), B(2) is moved into the B(3) location, and if we then find that A(3) is larger than B(1), A(3) is placed into the B(2) location. In this way, we keep going until we have examined, and placed into their proper places in the B array, all ten elements of the array A.

c) "Repeated Selection Sort": On the first pass, compare each A(I) with A(10), for I=1, 2, 3, ..., 9. At each comparison, interchange the values if A(I) exceeds A(10). On the second pass, compare A(I) for I=1, 2, ..., 8 with A(9). Keep going in this way until you are sure that the array A is properly ordered. Warning: Unlike the ripple sort, the absence of any interchanges at a certain pass of this process does *not* guarantee that the whole vector is ordered. Can you explain why?

2. *Merging:* Two vector arrays A and B, each of dimension ten, are to be read from data cards, five numbers to a card, and we want to produce a vector array C, of dimension twenty, containing all the elements of A and all the elements of B, as an ordered list. We assume that the separate vectors A and B are already in increasing order. Try to figure out an efficient method for doing this, by yourself. But then you might care to sneak a look at Chapter XII, where a merging operation is explained and coded up as a subprogram.

3. "*Merge-Sort*": This very useful technique is best explained by an example: Suppose the vector to be sorted has the ten components 2, 15, 7, 3, 5, 10, 9, 11, 20, 18. The three components 3, 5, and 10 form an increasing "string" within this vector, so do the three components 9, 11, and 20. The five separate "strings" (some of them only of length one) are: (2,15), (7), (3,5,10), (9,11,20), and (18). We reserve two more vector storage areas, B and C, say, each of dimension ten, and we place into B the first, third, and fifth strings, and into C the second and fourth (the even-numbered) strings. Thus B contains the elements 2, 15, 3, 5, 10, 18, b, b, b, b (where "b" stands for "blank" or not used locations within the array B); and C contains the elements 7, 9, 11, 20, b, b, b, b, b, b. Note that the contents of C happen to be already fully ordered, even though they started out as the two separate "strings" (7) and (9,11,20). We now merge the first string in B with the first string in C to produce a single ordered string in A; we merge the second string in B with the second string in C to produce a single ordered string further on in the A vector storage; and so on. When we run out of strings in either vector B or vector C, we just transfer the remainder of the other vector "as is" to the A locations. In our present example, there is only one merge operation, on the first string of B: (2,15) with the first, and only, string of C: (7,9,11,20). The result is the string (2,7,9,11,15,20) which is placed into the first six components of A. Since there is nothing more to be merged, the remaining components of B are placed straight into the remaining locations of A, so that A now reads: 2, 7, 9, 11, 15, 20, 3, 5, 10, 18. We now repeat the process: the odd-numbered strings of A (in fact, the first string (2,7,9,11,15,20)) are placed into B, the even-numbered strings of A (in fact, the second string (3,5,10,18)) are placed into C. We now merge the first (and only) string in B with the first (and only) string in C so as to get a completely sorted vector A: 2, 3, 5, 7, 9, 10, 11, 15, 18, 20. Write and debug a merge-sort program, for a vector with N components (assume N is no more than 100). *Warning: This is a particularly difficult exercise and should not be attempted right away.*

4. Improve the "vector version" of the average-of-marks program on page 110, as follows: Instead of insisting on exactly fifty students in the class, we replace the DO-coded loop starting with statement 1100, by an IF-coded loop. We read student card after student card, each time advancing NSTUD (the count of the number of students) by one, until we encounter a data card with the

number -1 as the first mark. This sends us to statement 1300, out of the loop. Note that the computation of the averages needs some revision, since we can no longer divide the sums by 50.0 and 200.0, respectively. Also, replace the STOP statement at the end by GO TO 1000, thereby making the program "recycling". As a final touch, add an echo-check for the contents of the student data cards, and arrange it so that the four marks of each student are preceded on the output page by the running number of the student within this class. The entire process is to stop when we see a data card with the number -1 as the mark in the *second* subject.

5. A fruit packer is shipping certain numbers of oranges, grapefruit, and tangerines in a single boxcar. These three numbers appear on the first data card. A second data card contains five real numbers, these being the prices per 100 oranges in each of five cities. The third data card contains the prices per 100 grapefruit, in the same five cities. And a fourth data card contains the prices per 100 tangerines. Store these data in appropriate vector arrays and matrix arrays (use a vector of dimension three for the numbers of the various fruits, and a matrix of dimension five-by-three for the prices). Then determine the five total prices fetched by the shipment if sent to each one of the five cities. Also determine the number of the city which yields the highest price for the shipment, and print out that number as well as the price fetched. Produce informative, well laid-out and documented (with messages) echo-check and result printout.

6. Same as problem (5), but include a fifth data card, containing the railway shipping charge for shipping one boxcar to each of the five cities. Allow for this in computing the packer's profit.

7. The Snotnose Club has five members. Not all members listen to each other. The five-by-five matrix array LISTN has matrix element LISTN(I,J) equal to one if member I listens to member J; otherwise, LISTN(I,J) equals zero. Not all members listen to themselves (i.e., LISTN(J,J) may be either zero or one), and the fact that I listens to J does not imply that J listens to I, i.e., LISTN(I,J) and LISTN(J,I) may be different from each other (the matrix array LISTN is not necessarily a "symmetric" matrix). Read in five data cards, each containing one *column* of the array LISTN. Echo-check. Then: A fire breaks out in the Club; member number "J" sees it and shouts "Fire". Determine, and print out, the number of members who perish in the fire because they do not listen to this club member. Do this in turn for J=1, 2, 3, 4, and 5.

8. After this disastrous fire, the Club accepts enough new members to bring the number up to five again, and institutes a new Club rule: When a member hears the cry "Fire", he must shout "Fire" himself, immediately. In the first "round of shouting", member J alone shouts, and certain members listen to this one shout. In the second round of shouting, these members (including member J himself if LISTN(J,J) is not zero) shout, and each club member hears KWAR(I,J) shouts of "Fire", this being the number of actual shouts he is prepared to listen to. Determine, and print out, the five values of KWAR(I,J) for I=1, 2, 3, 4, and 5, given some value of J; and do this in turn for J=1, 2, 3, 4, and 5. *Note:* These numbers KWAR can themselves be thought of as a five-by-five matrix array, and are in fact the matrix elements of the *square* of the matrix LISTN. The product of one matrix by another is defined on page 143, in problem 3; the square of a matrix is the product of a matrix with itself.

9. (More difficult) In later rounds of shouting, each man shouts once for every shout that he hears himself. Calculate the number of shouts from each man, in the third and fourth rounds (these are the cube and fourth power, respectively, of the matrix array LISTN).

10. (More difficult problem) Modify the work of problems 8 and 9 for the following reason: With the rules of the Club as stated so far, *all* members die in the fire, since they are obliged to stick around and shout "Fire" at each other, all the time, and none can escape. The new rule is: After a member has shouted "Fire" once, he is allowed to escape, so in the next round, he will not be around to do any shouting. Determine, for each J=1, 2, 3, 4, or 5 being the one to start the first round of shouting, how many members die in the fire.

CHAPTER XI

STILL MORE ABOUT INPUT AND OUTPUT: IMPLIED DO; TAPES AND DISKS

Section A: The Implied DO Loop.

So far, we have given only two methods for specifying output of arrays or their elements: (i) We may specify a particular array element explicitly, e.g., MARK(7,2); or (ii) We may specify the entire array, e.g., MARK. Both these methods have serious defects in practice. FORTRAN permits a more flexible form of specification, which is the subject of this section.

Suppose we have a vector array G, of dimension twenty-five, but we wish to print out array elements number 3, 5, 7, 9, 11, 13, and 15, only, not the other elements. One way to achieve this is to specify each array element in turn:

```
WRITE (NPRNT,1200) G(3),G(5),G(7),G(9),G(11),G(13),G(15)
```

As an alternative, FORTRAN allows us to write:

```
WRITE (NPRNT,1200) (G(K),K=3,15,2)
```

The variable "K" here acts exactly like the loop variable in the DO statement

```
DO ... K=3,15,2
```

That is, K equals 3 at the beginning, is then incremented by two (to 5), then again by two (to 7), and so on, until K reaches the termination test value 15. Thus the *implied DO loop* "(G(K),K=3,15,2)" specifies exactly the same array elements as the explicit list "G(3),G(5),G(7),G(9),G(11),G(13),G(15)".

More than one quantity may appear within such an implied DO loop. For example, let H be another vector. The statement

```
WRITE (NPRNT,1205) L,B,(G(K),H(K),K=1,7,2)
```

is equivalent to

```
WRITE (NPRNT,1205) L,B,G(1),H(1),G(3),H(3),G(5),H(5),G(7),H(7)
```

Notes:

1. The items in the list may or may not depend on the value of the loop variable. For example, the list (B,G(J),J=1,5,2) is equivalent to "B,G(1),B,G(3),B,G(5)", corresponding to the values of J in the DO loop "DO *n* J=1,5,2"; that is, J=1, then J=3, finally J=5. Since B does not depend on J at all, it is simply repeated three times.

2. The loop variable itself may appear in an output list. (*) For example, the list item

---

(*) Not in a READ list, though! The implied DO *specifies* values of J, so we can not *input* values of J at the same time!

may be as follows: (J,G(J),J=1,5,2). With this list, the current value of J is transmitted each time around the implied DO loop, just before the value of G(J).

3. We are allowed to put loops within loops, i.e., to organize nested loops. For example, let MARK be a matrix array, of dimension fifty-by-four, and let NUMBR be a vector array of dimension fifty (which contains identification numbers for the fifty students in the class). Consider the list item

   (NUMBR(I),(MARK(I,J),J=1,4),I=1,5,2)

   The inner implied loop is (MARK(I,J),J=1,4), which is equivalent to the expanded list

   MARK(I,1),MARK(I,2),MARK(I,3),MARK(I,4)

   This inner loop is enclosed in an outer loop on I, where I takes the values 1, 3, and 5, in turn. Thus the list as a whole is equivalent to the explicit list:

   NUMBR(1),MARK(1,1),MARK(1,2),MARK(1,3),MARK(1,4),
   NUMBR(3),MARK(3,1),MARK(3,2),MARK(3,3),MARK(3,4),
   NUMBR(5),MARK(5,1),MARK(5,2),MARK(5,3),MARK(5,4)

If we couple this kind of list with a suitable FORMAT statement, we can print out a whole table of values, in an easily readable sequence, and with table heading, by a single WRITE and FORMAT statement:

```
      WRITE (NPRNT,1300) (NUMBR(I),(MARK(I,J),J=1,4),I=1,50)
 1300 FORMAT (31H STUDENT MARKS IN FOUR SUBJECTS//12H STUDENT NO.,
     1 2X,4HPHYS,6X,4HCHEM,6X,4HMATH,7X,3HBIO//(I8,4I10))
```

This produces a table which might start as follows:

STUDENT MARKS IN FOUR SUBJECTS

STUDENT NO.	PHYS.	CHEM	MATH	BIO
34728	71	85	62	98
26457	23	45	15	68
.....	....	....	....	....

The table goes on for a total of fifty lines, each line containing the identification number of a student, followed by his marks on the four subjects. (If you have forgotten the rules about rescanning of FORMAT statements, refresh your memory by looking at page 78).

Section B: Reading and Writing Magnetic Tapes and Disks.

In addition to line printers, card punches, and card readers, modern computers come equipped with magnetic tapes, or magnetic disks, or magnetic drums, or several of these devices. These are "auxiliary" or "back-up" memory, ways for the machine to remember, and use in later computations, large masses of information which can not be kept in core memory.

In commercial data processing, tapes and disks are used to keep large "files", for example, a file of all customers of an insurance company, with their policies, premium rates, bonus rates, expiry dates, etc. Such files are "created" by making up a set of input cards, reading these cards into the machine core memory, one card at a time, and then

transferring from core memory to magnetic tape or disc. Subsequently, it is likely that the file will have to be "updated" to allow for new customers, etc.

In scientific computation, the data stored on a tape may come, for example, from a rocket flight. The radio signals sent back by the rocket are amplified electronically, converted into a form appropriate for storing onto computer tape, and transferred to this tape while the rocket is in flight. No cards are punched: The information goes straight onto tape. Once the tape has been written, the computer analysis of the data can take much more time than the actual rocket flight.

Magnetic tapes attached to computers are similar to those used in tape recorders, except much wider, more sturdy, and very much longer. A typical computer tape is half an inch wide and 2400 feet long. Information is stored in magnetized spots across the tape, seven spots in a line for "seven track tape", nine spots in a line for "nine track tape". There are between 200 and 2000 such lines of spots per inch length of the tape (depending upon the tape drive used). Thus the total memory capacity of a magnetic computer tape is very much larger than of usual core memories. Furthermore, reels of tape can be mounted and dismounted, labelled, and stored away as "private tapes" for future use. Hence the memory capacity of a machine with tape drives attached is unlimited in principle, though of course there are practical limits.

Magnetic "disks" are sets of large horizontal disks, mounted on one vertical spindle. Each disk has an upper and a lower surface, coated with magnetic material and thus capable of retaining information in magnetized spots. A set of "read-write heads" fits in between the various disks, in such a way that any spot on any disk can be read, or overwritten, by (i) moving the heads forward or back, and (ii) once the heads are in the right position, waiting until the normal rotation of the disks on their spindle has brought the desired spot close to one of the read-write heads.

Usually, but not always, magnetic disks can be dismounted and kept as private disks. But mounting and dismounting of disks is more cumbersome than for tapes, and disks are more expensive than tapes. As against this, the information on a disk is more easily accessible than the information on a tape. If one wants a particular piece of information located, say, near the center of the tape, one must wait while the tape winds forward, or back, to the desired spot. This can take many seconds, or even minutes. Disks have "direct access": one can, in a tenth of a second, get the information from any specified spot on the disk. But it must not be forgotten that a tenth of a second is still a long time at present computer speeds. A number can be fetched from main memory in about a millionth of a second!

Information on a magnetic tape appears in two different forms, an "internal" and an "external" form. Let us illustrate this with a specific example: Suppose that a "memory location" in main memory is sixteen binary digits long, and that these sixteen binary digits represent the integer 29. In binary form this number is written as

0000000000011101

and this is the way it appears in the memory of this particular machine. When this is written onto a tape in "internal" form, two rows of magnetic spots are written. The first row contains the first eight binary digits, the second row contains the remaining binary digits:

Row 1:	00000000	"Internal form"
Row 2:	00011101	

On "nine-track" tape, there is a ninth binary digit in each row of magnetic spots, but

that digit carries no additional information; rather, it serves as a checking digit.

When the same information is written in "external" form, a FORMAT field descriptor is required. Let us suppose that the field descriptor is I6. Then the binary number in the machine is converted to the six characters "blank", "blank", "blank", "blank", "2", and "9". These six characters appear (in coded form) as six rows of magnetic spots on the tape:

Row 1:	(coded blank)	
Row 2:	(coded blank)	
Row 3:	(coded blank)	"External form"
Row 4:	(coded blank)	
Row 5:	(coded "2")	
Row 6:	(coded "9")	

Clearly, the 'external' form is more cumbersome, takes longer to produce, and occupies more room on the tape. 'Internal' form is preferred for most purposes, but there are occasions (for example, when the same tape has to be processed by two different computers) when external form must be used.

Both in internal and in external form, quite a few numbers on the tape are lumped together into one "record". For example, the contents of seventy-five memory locations might be lumped together, in internal form, as one record consisting of 150 rows of magnetized spots on the tape, two rows for each of the seventy-five numbers. After this, there is an "inter-record gap", which is just a blank area on the tape. Then comes the next record; and so on.

Before reading or writing magnetic tape, the "read-write head" in the magnetic tape drive is positioned on some inter-record gap. The *next* record is then processed in response to the programmer's command (READ or WRITE, as the case may be). Thus, which record on the tape is processed next, depends entirely on the position of the tape at the moment. It is up to the programmer to keep track of where he is on the tape, and to make sure that the record he is telling the machine to read is the record he wants read, or that the record which he is telling the machine to write is being written where he wants it to be written (and not, perhaps, right on top of some other record which the programmer wanted to preserve!).

Records on tape, as well as inter-record gaps, are not always the same exact physical length. For example, the length of an inter-record gap may depend on the adjustment of the tape drive, and on the temperature in the machine room, at the time this tape was written. The result is that when we write a record onto the tape, then other records which appear later on that same tape become unfindable and can no longer be read. "Updating" of a magnetic tape therefore requires rewriting all the information on the old tape (including information which does not need alteration) onto a new tape, inserting whatever new records are wanted, or deleting whatever old records are no longer wanted, at the appropriate places. Updating a tape is therefore an expensive, time-consuming operation, and magnetic tapes are most useful if updating needs to be done only rarely. If frequent updating is required, then magnetic disk storage is better.

A large number of records on a magnetic tape are combined into one "file". There is an "end-of-file mark" between any two files. It is usual to keep only one file on any one tape. But occasionally, a file may be too long to fit onto a single tape, and must be spread over several tapes. And there are occasions when it is advantageous to keep several distinct files on one tape.

From the programming point of view, each file is assigned a "unit number" in FORTRAN programming. The monitor system must be told, in some way, that FORTRAN logical unit number 25, say, corresponds to a file of a certain type, written in a certain way, onto a certain device (tape, disk, etc.), with a certain label, and what have you. This is done by means of "system control cards" which are read and processed by the monitor system, not by the FORTRAN compiler.

At present, these system control cards are a major problem to programmers (and not just beginning programmers, either!). There is no standard language, such as FORTRAN, for system control cards. On the contrary, these cards differ from manufacturer to manufacturer, from machine to machine, and even from installation to installation with the same machine! Do not, repeat not, attempt to write your own system control cards. Ask an expert at your installation, then start praying. In this author's opinion, standardization and simplification of the "job control language" is the very highest order of priority in present-day computing practice. The complete chaos which exists at the moment is as intolerable as it is inexcusable.

The kind of file which we have been describing, and which is the only kind possible on magnetic tapes, has record after record in sequence. It is called a "sequential file". USA Standard FORTRAN IV includes commands for handling such sequential files, and these commands will be given shortly. On disks, but not on tapes, a different kind of file organization is possible, called a "direct-access file". Each record in a direct-access file is assigned a record number, and the machine can be instructed directly to go to record number so-and-so on direct-access file number such-and-such. Many manufacturers provide "direct-access commands" in "their" FORTRAN. But these commands are not yet part of USA Standard FORTRAN IV, and are therefore not discussed in this Chapter. Direct-access commands for your version of FORTRAN are presented briefly in Chapter XVII. The advantage of direct-access files is that it is not necessary to space forward through record after record which we do not want, merely to get to the record which we do want.

This is quite an advantage, but this advantage should not be overrated. By careful planning of the program from the start, it is usually possible to write information on a tape in the same sequence in which we shall wish to read that information later on. If this is done, the tape can be read forward sequentially, record after record, and this can be done in "buffered" fashion, meaning that the machine is carrying out computations on the information from one record, at the same time that the information from the next record on tape is being transferred to a "buffer area" in main memory. If computation on the numbers in one record takes longer than reading one record (this is usual if the records are sufficiently long), then almost no time is lost by reading tape. It will still be necessary to "rewind" the tape back to its starting point after each complete computation. But rewinding is much faster than reading record after record. Furthermore, it is possible to write tapes in duplicate, and read them in tandem: While one tape is being rewound, the other tape is being read; and vice versa. Of course, the methods employed for the computation have to allow for the nature of tapes. For example, the ripple sort (page 114) is entirely unsuitable for sorting a set of numbers stored on a tape; whereas the merge sort (page 115) is well suited to sorting a sequential file.

Although the monitor system does not permit, on a sequential file, to write a FORTRAN command to access record number such-and-such, there is nothing to prevent a programmer from doing the same thing by sensible programming. For example, the first number stored on each record may be (and we advise that it should be) a sequential record number. This record number is written as part of the record itself, when the tape

file is produced. Later on, when we read that record, the first number read into main memory is the record number. This record number can be, and should be, compared in the machine with the record number the program expects to be reading at this moment. Since sequential files usually contain large numbers of similar records, this programming precaution is strongly recommended.

We now present the FORTRAN commands for reading, writing, and manipulating sequential files.

To write one record in "internal" form, the FORTRAN command is

WRITE (*n*) *List*

where *n* is the unit number assigned to that file, and *List* is a standard output list. Numbers are fetched from the memory locations specified in the list, one by one, and are written onto the file in "internal" form. When the list has been exhausted, an inter-record gap is written. Note that this WRITE command does not specify, or require, an associated FORMAT statement. This is due to the fact that no conversion of the information is required: the information is written onto the file exactly as it appears inside the main memory of the machine.

To read a record which has been written in that way, the FORTRAN command is

READ (*n*) *List*

Information is taken from the file and stored into one after another of the memory locations specified in the list. If the size of the list is smaller than the size of the record on tape (in particular, if the list is missing altogether, which is permitted), then we read the required numbers from the record, and thereafter space the tape forward to the next inter-record gap. If the size of the list exceeds the record size, an error is diagnosed, and execution of the program is terminated.

Note: The command READ (*n*) has the effect of spacing sequential file number *n* forward by one record.

In order to space the file (the tape) backward by one record, the FORTRAN command is BACKSPACE *n*.

In order to return sequential file number *n* to the beginning of its very first record, the FORTRAN command is REWIND *n*.

After writing the last record of a sequential file, the end-of-file mark is produced by the FORTRAN command END FILE *n*.

In order to write records on a sequential file in "external" form, the FORTRAN command is

WRITE (*n*,*m*) *List*

where *n* is the unit number of the file and *m* is the statement number of a FORMAT statement. This FORMAT statement controls the conversion from machine form to external (character) form, and also controls the number of records that is written: each slash "/" in the FORMAT statement produces one inter-record gap.

In order to read records in "external" form, the FORTRAN command is

READ (*n*,*m*) *List*

The FORTRAN commands BACKSPACE *n*, REWIND *n*, and END FILE *n* can be used with files in either form. Any one file must be purely in one form, however.

Summary and Formal Definitions for Chapter XI.

1. *Implied DO Loop:*

   The general form of an implied DO loop list item in an input/output list is one of:

   (*List,J=k,l,m*) or (*List,J=k,l*)

   where *List* is any valid input/output list, *J* is the name of an integer variable. The rules for the loop control variables *k, l,* and *m* are exactly the same as for the corresponding quantities in the DO loop: DO *n J=k,l,m*. The list is transmitted as often, and with exactly the same values of the loop variable *J*, as such a DO loop would be traversed. If the list is an output list, the loop variable *J* can itself appear within the list. In any case, the list may contain implied DO loop list items within itself.

2. *Reading and Writing Magnetic Tapes and Disks:*

   Numbers can be written in either "internal" form or "external". Groups of numbers form a "record". Inter-record gaps appear between records. Groups of records form a "file". A file is terminated by an "end-of-file mark" if it is a sequential file (but not if it is a direct-access file). "Direct-access files" can not be placed on tapes (they can be placed on disks), and USA Standard FORTRAN IV includes no direct-access file commands at present. "Sequential files" can be written and read using USA Standard FORTRAN IV. Each such file must be assigned a "unit number". The sequential file commands for the file with unit number *n* are:

WRITE (*n*) *List*	READ (*n*) *List*		"Internal form"
WRITE (*n,m*) *List*	READ (*n,m*) *List*		"External form"
BACKSPACE *n*	END FILE *n*	REWIND *n*	Either form

   where *m* in the second line is the statement number of a FORMAT statement.

   Recommendations: Internal form is usually preferable. Long records on tape are preferable to short records. The first item of the List should be a consecutive record number, to identify the record. Records should be written in the same sequence in which they are to be read afterwards. Writing tapes in duplicate is a good precaution against tape failures.

DRILL EXERCISES FOR CHAPTER XI.

1. Construct the WRITE and FORMAT statements for the following: The matrix B, of dimension 5-by-6, is to be printed out row by row, each row preceded by a row number. The table is to have a heading, saying MATRIX ARRAY B in one line, then a blank line, then a line containing the headings "ROW NO", "COLUMN 1", "COLUMN 2", ..., "COLUMN 6", all in appropriate places. The matrix elements are to be printed out to five significant digits.

2. Print out the diagonal elements (subscripts I and J equal) of the matrix array F, dimension 50-by-50, each element preceded by its subscript I. Provide suitable heading information.

3. State which numbers, in which sequence, and in what form of printout, will be printed as a result of:

   a)
```
      DIMENSION MACK(20),JACK(20,3)
      WRITE (NPRNT,1200) (I,MACK(I),(JACK(I,J),J=1,3),I=1,10)
1200  FORMAT (I4,I10,5X,3I8)
```

   b)
```
      DIMENSION MUTT(10,20),JEFF(20,10)
      WRITE (NPRNT,1300) (K,(MUTT(K,L),JEFF(L,K),L=4,18,2),K=3,10,5)
1300  FORMAT (9I10/10X,8I10/)
```

4. Construct statements for printing out the odd-numbered rows of the matrix array C, dimension 9-by-9. Each row is to be preceded by the row number, and a suitable heading is to be provided.

5. Print out the even-numbered rows of the matrix G, dimension 24-by-24, starting with row N and ending with row M (assume N and M are even, and M is bigger than N). Assume a line width of 120 characters for the printer. Print each row as a set of four lines, six matrix elements to a line. The first line contains the row number, the others are indented; a heading is provided also.

6. Print out those elements of matrix array H, dimension 25-by-25, which lie on and below the main diagonal (i.e., for which I in H(I,J) is greater than or equal to J). The print-out is to appear row by row, with each complete row preceded by the row subscript. Watch out for too long lines! (Hint: This problem can not be done entirely by means of implied DO loops in a single WRITE statement; use an outer DO loop on the row subscript I, instead).

7. Refer back to Chapter X, drill exercise 7 (page 113), for the definition of the checker board array MORDE(I,J).
   a) A data card has been prepared, containing sixty-four integers next to each other, in columns 1 to 64 of the card. The code for these integers is: "0" for a black king, "1" for a black piece, "2" for an empty square, "3" for a white piece, and "4" for a white king. Columns 1 to 8 of the card contain the codes for the first *row* of the board, columns 9 to 16 contain the codes for the second row, and so on. Produce a READ and FORMAT statement for reading this card into the array MORDE, and follow this with commands to change the numbers so stored into MORDE, to the standard convention of exercise X.7;
   b) We wish to punch out a card with the present position of the checker board, according to the specifications in (a).

8. A certain long job takes many hours on the computer. To ensure against a machine failure during the run, "restart information" is to be punched out onto cards, every ten steps. This restart information consists of a 15-by-15 matrix array D. Write a program for punching out this matrix whenever an integer variable called I has a value divisible exactly by ten. The first seventy-two columns of each card are to be used as efficiently as possible (minimize the number of blank columns); but we require E-format for all real numbers, with seven significant digits for the mantissa. Columns seventy-three to eighty are used for card identification: They contain the characters "SCR", followed by the value of I/10 as a three-digit integer, followed in turn by a two-digit card sequence number.

9. A matrix array V has five rows and eight columns. Data cards contain two numbers each, which are to be read into columns five and six of the array, one pair of numbers for each row.

10. The matrix array YAHOO has dimension fifteen by nineteen. Data cards containing seven numbers each are to be read, and the numbers stored sequentially into the even-numbered rows of YAHOO, row after row.

11. Write a program segment for reading in the cards punched out in problem 8. Each input card is to be checked carefully. We want to ensure that all the cards belong to the same value of I, and that the cards are in proper sequence. An error message is to appear if these conditions are violated. This error message should give a card image of the faulty card.

12. Same as problem 8, but the restart information is to appear as one record on a tape, which has been assigned the logical unit number eight. The information is to appear in "internal" form (this excludes character information, such as "SCR"). A message should be printed out after each checkpoint (restart) record has been written onto the tape.

13. Write a program sequence for a program restart using the tape of problem twelve. We rewind the tape, then we read a data card with the value of I at which we want to restart. We check that I is exactly divisible by ten. If it is, we space forward on the tape through the correct number of records to get to the record for checkpoint I. We read that record in full, storing the information into the relevant places.

14. A tape, with logical unit number seven, contains coordinates "x", "y", and "z" of a rocket at successive times, 0.01 seconds apart. The last set of coordinates has z negative, indicating that the rocket has dived into the sea (sea-level is considered equal to z=0). Write a program to compute the total distance travelled by the rocket along its curved path (replacing the actual curved path by a sequence of straight line segments). The information on the tape is stored in internal form, with the record number NREC and the values of X, Y, Z at one time forming one record.

15. Same as 14, but the records are "blocked" for better efficiency of utilization of the tape. Each record contains the record number NREC, followed by twenty sets of values of X, Y, Z, at twenty consecutive times. The last record has zeros for X, Y, Z after the point at which the rocket has dived into the ocean. (*Note:* Short records waste valuable tape space with record gaps. Monitor systems often permit blocking of records with monitor control cards. Avoid this! Block it yourself!)

CHAPTER XII

SUBPROGRAMS

Section A: The FUNCTION Subprogram.

Experienced programmers break up the task of writing a program, by organizing the program into a number of separate and distinct "program units". One of these program units (only one) is the "main program". All the other program units are called "subprograms". The various program units can be, and often are, written by different people, compiled at different times, debugged separately, and put together into one big program only later on. When the whole program is put together ("linked"), program execution starts with the first executable statement of the main program. At various places within the main program, subprograms are "invoked". Each time a subprogram is invoked, control is transferred to that region of machine memory where this particular subprogram is stored. The orders within that subprogram are executed by the machine; and at the end, control is returned back to the main program. Furthermore, each subprogram may itself invoke other subprograms, and so on. In each case, control is returned to whichever program unit was responsible for the invocation, at the appropriate point in that program unit.

Complicated though this may sound, it is good sense to organize programs in this way. The longer a program unit is, the harder it is to debug, and the more time is wasted on recompilations during the debugging stage. Furthermore, a subprogram, once written and debugged, can be used in a number of different applications. It becomes, in effect, an extension of the "memory" of the machine.

We start with an example of a FUNCTION subprogram. The function ROOT(X,N) is intended to be the n'th root of the absolute value of x. The FORTRAN commands for doing the actual computation are:

```
Y=ABS(X)
EN=N
ROOT=Y**(1.0/EN)
```

where the second command was necessary to convert the integer N to the real number form EN.

Let us now give the FORTRAN subprogram which *defines* the function ROOT(X,N):

```
FUNCTION ROOT(X,N)
Y=ABS(X)
EN=N
ROOT=Y**(1.0/EN)
RETURN
END
```

and we shall now discuss and explain these FORTRAN statements.

1. The first statement informs the compiler that what follows is a subprogram (rather than part of the main program); that this subprogram is of the FUNCTION type (see Section B for subprograms of SUBROUTINE type); that the *name* of the function is "ROOT", and that the function *value* is a real number rather than an integer (because the name ROOT is a real-variable name in FORTRAN); that this function depends on two separate *dummy arguments,* the first of which is a real number, the second of which is an integer. The particular names (X and N in our case) given to the two dummy arguments do not matter at all, since the dummy arguments are in any case replaced by actual arguments whenever the subprogram is invoked (see later). What does matter is that X is a real-variable name, and N is an integer variable name.
2. The next three statements are the "meat" of the subprogram. These statements tell us how to compute the function value ROOT from the values of the arguments.
3. The statement RETURN is an instruction to *return control* to the superior program unit which was responsible for invoking this subprogram.
4. The final statement, END, tells the compiler that we have come to the end of the FORTRAN cards for this particular subprogram. Whatever other FORTRAN cards there are in the deck, have nothing to do with defining the FUNCTION ROOT(X,N).

Note that we have used the name of a built-in function, ABS, within this subprogram. Not only is this permissible, but we can use the names of whatever other subprograms are available, either as built-in functions, or supplied by ourselves or other programmers. The only subprogram we must *not* invoke is ROOT, since we can not (within FORTRAN) define a function in terms of itself.

Now suppose that we want to evaluate, within the main program, the sum of the fifth root of A and the fourth root of B, and store the result into memory location C. The FORTRAN statement which achieves this within the main program reads:

```
C=ROOT(A,5)+ROOT(B,4)
```

This statement results in two separate invocations of the subprogram ROOT. The first time, the dummy argument X is replaced by the actual argument A, the dummy argument N is replaced by the number 5, and the value of ROOT is computed to be the fifth root of A. The second time, the dummy argument X is replaced by B, the dummy argument N is replaced by 4, and the value of ROOT is computed to be the fourth root of B. The sum of these two values is then stored into memory location C.

Once the subprogram ROOT has been defined, it can be used in quite complicated expressions. For example, suppose that we want the cube root of $1+(r^2+1)^{1/2}$ to be stored into memory location Q. The FORTRAN statement reads:

```
Q=ROOT(1.0+ROOT(R**2+1.0,2),3)
```

The sequence of evaluation of this expression is as follows:

1. Evaluate R**2+1.0, and store it into a temporary location, say W;
2. Invoke ROOT with the actual arguments W and 2, getting the result $(r^2+1)^{1/2}$;
3. Add 1.0 to this result, and store the sum in another temporary location, say V;
4. Invoke ROOT once more, this time with the actual arguments V and 3;
5. Store the final result into memory location Q.

The *dummy arguments* on which a function depends must be identifier names, not constants or complicated arithmetic expressions. But each *actual argument*, used

in the calling (superior) program unit which invokes the function ROOT, can be anything, as long as it is of the same type (real or integer) as the type declared for the corresponding dummy argument.

Besides dummy arguments which are identifier names of single memory locations, we are also allowed *dummy array names.* In order to declare that a certain dummy argument is the name of an array, we must place into the subprogram a DIMENSION statement for an array by that name. When the function subprogram is invoked elsewhere, the actual argument must also be an array name, and the array must be declared, in the calling program, to have exactly the same dimensions.

A FUNCTION subprogram is not permitted to alter the arguments of the function. Thus, the evaluation of FUNC(X) must not result in alteration of the value of X. (This rule is specific to Basic FORTRAN IV; it is relaxed in full FORTRAN IV.)

In order to illustrate the usefulness of FUNCTION subprograms, as well as of arrays as arguments, we shall now rewrite the binary search through an ordered list (Chapter X, page 106) as such a subprogram. We shall use the language of Chapter X, that is, we shall refer to taxable income, tax brackets, etc.; but the concepts, and the FUNCTION subprogram based upon these concepts, are not restricted to this special application. Any ordered list can be searched in this way.

The *value* of the function is the number of the tax bracket, which is an integer. We shall therefore give an integer name to the function, call it INDEX.

The *arguments* on which the function value depends are:

1. The value of the taxable income, call it X;
2. The array name of the array which contains the ordered list of the tax brackets; in our case, this is a vector array of dimension twenty-two. Let us call it OLIST (for "ordered list");
3. The number of actual entries in that list, which in our case is also equal to twenty-two. But we want to allow for the more general case that the dimension of the vector is larger than the number of actual elements of the ordered list. This is absolutely necessary if the list requires updating from time to time, as most lists do. Let us call the number of actual list elements N. We shall assume that the value of N is always less than or equal to twenty-two.

Except for the substitution of new names (INDEX instead of KRACK, OLIST instead of BRACK) the flow diagram is exactly the same as before (page 106), and so is the code. We therefore give only the FORTRAN code itself, revised now as a FUNCTION subprogram, rather than as a main program. This should be compared with the FORTRAN code on page 106.

```
      FUNCTION INDEX(X,OLIST,N)
C     THIS FINDS THE LOCATION OF THE NUMBER X WITHIN THE
C     ORDERED LIST OLIST, CONTAINING N LIST ITEMS IN STRICTLY
C     INCREASING ORDER.
C     INDEX = 0 IF X IS LESS THAN OLIST(1)
C     INDEX = K IF OLIST(K) LESS THAN X, AND X LESS THAN OR
C                 EQUAL TO OLIST(K+1)
C     INDEX = N IF X IS GREATER THAN OLIST(N)
C     METHOD USED = BINARY SEARCH, ASSUMING NO DUPLICATE
C                   ITEMS EXIST IN THE LIST.
C
```

```
      DIMENSION OLIST(22)
C     FIRST CHECK WHETHER X IS INSIDE THE LIST AT ALL
 2000 IF(X-OLIST(1)) 2100,2100,2300
C     X LESS THAN OR EQUAL TO OLIST(1)
 2100 INDEX=0
      RETURN
C     TEST OTHER END OF LIST
 2300 IF(X-OLIST(N)) 2600,2400,2500
C     X EQUALS OLIST(N)
 2400 INDEX=N-1
      RETURN
C     X EXCEEDS OLIST(N)
 2500 INDEX=N
      RETURN
C     PREPARE BINARY SEARCH
 2600 IGH=N
      LOW=1
C     LOOP STARTS HERE
 2800 MID=(IGH+LOW)/2
 3000 IF(X-OLIST(MID)) 3100,3050,3200
C     X EQUALS OLIST(MID)
 3050 INDEX=MID-1
      RETURN
C     X LESS THAN OLIST(MID)
 3100 IGH=MID
      GO TO 3300
C     X GREATER THAN OLIST(MID)
 3200 LOW=MID
C     NOW TEST WHETHER WE HAVE IT BRACKETED TIGHTLY
 3300 IF(IGH-LOW-1) 5000,5000,2800
C     SEARCH IS FINISHED
 5000 INDEX=LOW
 5200 RETURN
      END
```

Notes:

1. Notice the large number of comment cards at the beginning of this subprogram. This is most desirable if the subprogram is to be used by anyone else, and even if it is going to be used by yourself only, at some time in the future. It is surprising how easily one forgets just precisely what a program does; and how difficult it is to find out by reading the FORTRAN program. "Documentation" of "software" (that is: standard programs for a computer) is essential.

2. There are a number of RETURN statements in this subprogram. Since each RETURN statement is translated into quite a few machine commands, it would be better to replace all but the last RETURN statement by: GO TO 5200.

3. The subprogram *assumes* that OLIST is an ordered list, and that N is a positive number; failure will occur otherwise. In a well-designed standard subprogram, it is not permissible to just "assume" such things. Rather, whatever can be checked on the spot, should be so checked (for example, it would not be hard to insert a test to see whether N is indeed a positive integer).

4. The DIMENSION statement for OLIST assigns this vector array a dimension of twenty-two. Taking the FORTRAN language rules literally, one would have to

conclude that FUNCTION INDEX can be used only if the ordered list has been assigned this particular dimension in the calling program. This is a severe, indeed unacceptable, restriction. There are two ways out of this:

a) In full FORTRAN IV, as opposed to Basic FORTRAN IV, a subprogram can specify an "adjustable dimension" for an array dummy variable, provided that the value of that dimension is itself a dummy argument.

b) Even in Basic FORTRAN IV, however, most compilers do not really object if the rule about dimensions is violated for *vector* arrays. (Only special student compilers are that finicky.) Most compilers permit DIMENSION OLIST(1) in the *subprogram.* The actual dimension is the one appearing in the superior program. But this trick does *not* work for *matrix* arrays.

All right, now we have *defined* the FUNCTION subprogram INDEX. How do we go about *using* this subprogram?

Well, suppose we want to do exactly what we did in Chapter X, that is, read data cards for the tax brackets, then read values of the taxable income TINC, and for each such value, determine the tax bracket into which this income belongs. The main program for doing so consists essentially of the outer portions of the FORTRAN program on page 104 and 106; the inner portions (statements 2500 to 5000, inclusive) are now replaced by a single statement, invoking the FUNCTION subprogram. Here is the main program:

```
C       MAIN PROGRAM FOR DETERMINING TAX BRACKETS
        DIMENSION BRACK(22)
        NREAD=1
        NPRNT=3
C       READ THE TAX BRACKETS
 1000   READ (NREAD,1010) BRACK
 1010   FORMAT (4E10.0)
        WRITE (NPRNT,1020) BRACK
 1020   FORMAT (13H TAX BRACKETS  /(4F20.2))
C       READ DATA CARD WITH TAXABLE INCOME
 2000   READ (NREAD,1010) TINC
        IF(TINC) 2100,2100,2500
 2100   STOP
C       SET UP SEARCH BY INVOKING SUBPROGRAM INDEX
 2500   KRACK=INDEX(TINC,BRACK,22)
        WRITE (NPRNT,5100) TINC,KRACK
 5100   FORMAT (9H INCOME =,F17.2,10X,13HTAX BRACKET =,I4)
        GO TO 2000
        END
```

Notes:

1. The actual arguments with which INDEX is invoked are: TINC, BRACK, and 22. The name TINC differs from the name X of the dummy argument, but this is perfectly all right. What matters is that both TINC and X are quantities of type real (not integer), and that both are the names of single variables (not arrays). Similarly, the name BRACK differs from the dummy name OLIST; but both are the names of arrays, more precisely, of vector arrays of dimension twenty-two. Finally, the last actual argument is an integer constant (22), whereas the last dummy argument was an integer variable name, N. But both are of type integer.

2. Since the identifier name INDEX is used in statement 2500 as the name of a FUNCTION subprogram, this name can not be used for anything else within the main

program. For example, a statement such as INDEX=INDEX(TINC,BRACK,22) would be grammatically incorrect, since (on the left of the equal sign) there is an attempt to use "INDEX" as the name of a memory location into which a number can be stored.

3. On the other hand, names which appear only *within* the subprogram are "local" to this subprogram, and can be used with a different meaning, if desired, in the main program or in some other subprogram. For example, it would be perfectly correct to employ the identifier name MID somewhere in the main program, as meaning something quite different from, and entirely unrelated to, the name MID in the subprogram. Conversely, the fact that a memory location by the name of MID has been set to some definite value within the subprogram, does not allow us to fetch that particular value from a memory location with name MID in the main program (this is a very common mistake of beginners). The name "MID" refers to one memory location if it is used in the subprogram; it refers to quite a different memory location if it is used in the main program. The two have nothing to do with each other, and putting some value into one, does not do anything to the other.

4. Similarly, statement numbers within the subprogram have no relation to statement numbers in the main program. For example, there is a statement number 2500 in the subprogram, and also a statement number 2500 in the main program. This does not violate the rule about "no duplicate statement numbers". The rule is that *there must not be duplicate statement numbers within the same program unit.*

5. We see that there is a "division of labour" between the main program and the subprogram. The main program is devoted to reading of data, and printing out of results. The "hard work" is done by invoking the subprogram INDEX (through statement number 2500 of the main program). It is the subprogram which performs the actual search through the list. Standard subprograms, like this list search program, should not do any reading of data or printing of output (except output of an error message in case of a failure). The purpose of FUNCTION INDEX is to find one number, the order number within the list. It should stick to that purpose.

All right, so now we have a main program (page 129) and a FUNCTION subprogram (pages 127-128), and we intend to employ the same data cards as before (page 104). How do we put all this together into one "machine job"?

The details differ from machine to machine, and are presented for your machine in Chapter XV. But certain general principles apply in all cases. These are shown in a schematic fashion, in the diagram below:

Order in Job Deck	*Type of Material*	*Comments*
1	Job Card	Contains programmer name etc.
2	System Control Cards	To invoke FORTRAN compiler etc., not needed in some systems.
3 (*)	FORTRAN cards (subprogram)	See pages 127-128; first card is FUNCTION statement, last card is END statement.
4	More system control cards	In some, deplorable, systems the FORTRAN compiler must be re-invoked here! Horrid.
5 (*)	FORTRAN (main program)	See page 129; END card is last.
6	More system control cards	Only one is necessary here in a good system.
7	Data Cards	See bottom of page 104.
8	More system control cards	Still more excrescences, in some systems!

At this point, we can finally appreciate the difference between the FORTRAN card END, and the system control card signalling the start of the execution phase (item 6 in the description of the job deck). "END" tells the compiler that we have come to the end of the FORTRAN cards for one program unit; the last card of every FORTRAN program unit must be an END card. The system control card of item 6 tells the monitor system that we have arrived at the end of the entire task of translation from FORTRAN to machine language. These pieces of information are logically quite different, and become one and the same if and only if there are no subprograms at all. Only then can there be any misunderstanding about the difference.

In many systems, it is possible to instruct the compiler to produce punched cards containing the translated version of each FORTRAN program unit. These cards form what is called an object deck. In subsequent runs with the same program, but different data cards, it is possible to bypass the compile step altogether, by using the object decks directly. As an alternative to producing object decks in card form, the "card images" of the object deck cards may be stored on a disk or tape, by appropriate use of system control cards. However, student compilers do not usually give such options, since most student runs are of the "compile-and-go" type anyway.

Section B: The SUBROUTINE Subprogram.

The FORTRAN language provides a second type of subprogram, called the SUBROUTINE subprogram, for doing things which can not be done with a FUNCTION subprogram. FUNCTION subprograms are not permitted to alter the arguments of the function. Suppose that we want to sort an unordered list ULIST, of length N, so that it is rearranged as an ordered list (items in increasing size along the list). Then we must alter the sequence of items in the original list. This can not be done with a FUNCTION subprogram; but it can be done with a SUBROUTINE SORT(ULIST,N).

The main differences between a SUBROUTINE and a FUNCTION subprogram are:

1. A FUNCTION subprogram returns its result as a function value; within the subprogram, we must set a location named with the name of the function (for example, see statement number 5000 on page 128). By contrast, the name of a SUBROUTINE subprogram is just a "name to remember by", and has no other meaning. Instead, the SUBROUTINE subprogram returns its result, or results, by means of altered values of one or more of its arguments.

2. The method of invocation is different. A FUNCTION subprogram is invoked by using the function name within an arithmetic expression. A SUBROUTINE subprogram is invoked by means of a special kind of statement in the superior program, namely the "CALL" statement. For example, SUBROUTINE HEAVE(HO) may be invoked in a superior program by the statement

   CALL HEAVE(HOIST)

   HOIST is then the actual argument which replaces the dummy argument HO, and an altered value of HOIST may be returned to, and used by, the superior program. A SUBROUTINE subprogram may, but need not, have dummy (and actual) arguments.

(*) (Footnote for page 130) In some systems, the main program must be compiled after all the subprograms (as shown in the diagram). In other, equally deplorable, systems, the main program must be compiled first, before all the subprograms. In good systems, program units can be compiled in an arbitrary sequence.

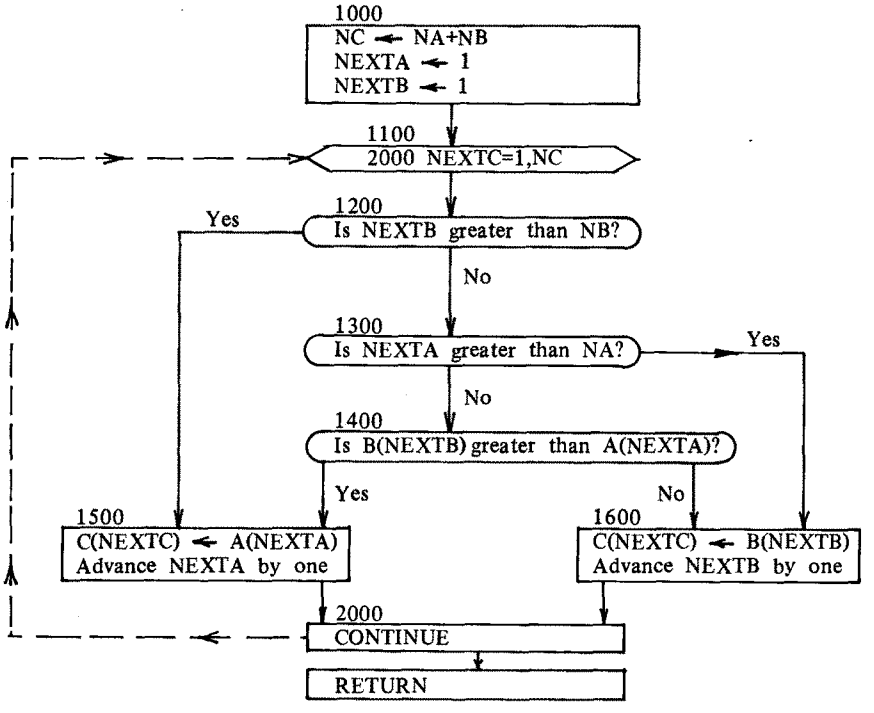
As an illustration of a SUBROUTINE program, let us consider the following: In data processing, we frequently need to merge two ordered lists into one combined, ordered list. For example, we may have a list of NA elements, already ordered, stored in the vector array A of dimension 100 (the actual number of list elements, NA, can be anything from 1 to 100). We may have another ordered list, of NB elements, stored in the vector array B also of dimension 100. We wish to produce a merged, ordered list of NC=NA+NB elements, stored into the first NC array elements of the vector array C, of dimension 200.

Let us write a SUBROUTINE MERGE(A,NA,B,NB,C,NC) for performing this operation. We first discuss what is to be done. Then we give a flow diagram, and finally the FORTRAN code.

At any given moment, during the merging process, we know that the number to be used next is either A(NEXTA) or B(NEXTB), whichever is the smaller. This number is to be stored into location C(NEXTC), where NEXTC goes through the values 1, 2, 3, 4, ..., NC. We therefore compare A(NEXTA) with B(NEXTB). If A(NEXTA) is the smaller one, then we store it into C(NEXTC) and we increase NEXTA by one; if on the other hand B(NEXTB) is the smaller one, then we store *it* into C(NEXTC) and we increase NEXTB by one. In either case, we increase NEXTC by one (this is done by means of a DO loop on NEXTC, going from 1 to NC). We then repeat the process.

One caution must be observed, however. It may happen that we have stored away all the elements of the list A already, so that NEXTA is now equal to NA+1. In that case, we know without further comparison that all future numbers have to come from array B. Similarly, if at any stage NEXTB is already equal to NB+1, then array B has been exhausted, and all future numbers have to be drawn from array A. Thus, tests on the values of NEXTA and NEXTB must be performed before the comparison of A(NEXTA) with B(NEXTB).

We now present the flow diagram:



```
      SUBROUTINE MERGE(A,NA,B,NB,C,NC)
C     MERGING OF TWO ORDERED LISTS
C     LIST A HAS NA ITEMS, IN INCREASING ORDER
C     LIST B HAS NB ITEMS, IN INCREASING ORDER
C     RESULT LIST C HAS NC=NA+NB ITEMS, IN INCREASING ORDER
C
      DIMENSION A(100),B(100),C(200)
 1000 NC=NA+NB
      NEXTA=1
      NEXTB=1
 1100 DO 2000 NEXTC=1,NC
 1200 IF(NEXTB-NB) 1300,1300,1500
 1300 IF(NEXTA-NA) 1400,1400,1600
 1400 IF(B(NEXTB)-A(NEXTA)) 1600,1600,1500
 1500 C(NEXTC)=A(NEXTA)
      NEXTA=NEXTA+1
      GO TO 2000
 1600 C(NEXTC)=B(NEXTB)
      NEXTB=NEXTB+1
 2000 CONTINUE
      RETURN
      END
```

Notes:

1. Observe the detailed comments at the beginning of this subprogram. The importance of proper documentation of a subprogram simply can not be overemphasized. Documentation makes the difference between a useful subprogram, and a random collection of scrap computer cards.

2. Since A, B, and C are the array names of vector arrays, a DIMENSION statement must appear for these names. With most FORTRAN compilers, the actual numbers (100, 100, and 200) do not matter, and the subprogram will work quite generally. Only student compilers insist on the equality of the dimensions of actual and dummy arguments which are vector (rather than matrix) arrays.

3. We do *not* check that NA is less than or equal to 100, or greater than zero. The first bit of checking would make the program less general than it actually is; and the second bit of checking is unnecessary, since the subroutine works correctly even if the vector array A has no elements at all (even if NA equals zero). You should convince yourself of this, by looking at the flow diagram.

4. Merging cannot be done by a FUNCTION subprogram, since statements 1500 and 1600 both alter the array elements of array C, which is one of the dummy arguments.

Other standard subprograms can be written for handling lists. For example, programming exercise 1 at the end of this Chapter asks you to write a SUBROUTINE SORT which sorts the first N elements of the vector array ULIST so that this unordered list becomes an ordered list (elements rearranged in increasing sequence).

Let us now show how one makes use of such subroutines. Let us suppose that we want to read in a set of NA numbers, store them into array ALIST, followed by a set of NB numbers, to be stored into array BLIST. We want to sort both sets and print them out in proper sequence. Then we want to merge these two (now properly ordered) lists into one combined, ordered list CLIST, and we want to print out that final list.

The main program for all this is surprisingly short now:

```
C       MAIN PROGRAM FOR SORTING AND MERGING
C       THIS REQUIRES SUBROUTINES MERGE(A,NA,B,NB,C,NC)
C                                 AND     SORT(ULIST,N)
        DIMENSION ALIST(100),BLIST(100),CLIST(200)
        NREAD=1
        NPRNT=3
 1000   READ (NREAD,1010) NA,(ALIST(I),I=1,NA)
 1010   FORMAT (I5/(4E10.0))
        READ (NREAD,1010) NB,(BLIST(I),I=1,NB)
        CALL SORT(ALIST,NA)
        CALL SORT(BLIST,NB)
        WRITE (NPRNT,1020) NA,(ALIST(I),I=1,NA)
 1020   FORMAT (14H INPUT LIST OF,I5,19H  ELEMENTS, ORDERED/(7E16.6))
        WRITE (NPRNT,1020) NB,(BLIST(I),I=1,NB)
        CALL MERGE(ALIST,NA,BLIST,NB,CLIST,NC)
        WRITE (NPRNT,1030) NC,(CLIST(I),I=1,NC)
 1030   FORMAT (14H FINAL LIST OF,I5,19H  ELEMENTS, ORDERED/(7E16.6))
        STOP
        END
```

Notes:

1. The main program does not do any "real work". For that, it calls upon the subroutine subprograms SORT and MERGE. The main program sets aside storage for the various arrays (the numbers "100" and "200" in the DIMENSION statement *do* matter here, very much, since *this* is where the actual storage is reserved: in the main program, where the array name is an actual argument, not in the subprogram, where the array name is only a dummy argument). The main program then inputs numbers, and prints out results.

2. In order to keep things simple, we have written this main program without the checking of input numbers which it really ought to do. We should check that NA lies between 1 and 100, inclusive, ditto for NB. Also, it would be better to provide an honest echo-check of the input numbers, i.e., to print out the array ALIST before entering SUBROUTINE SORT, as well as afterwards; ditto for BLIST.

3. If the lists ALIST and BLIST are at all lengthy, a different form of printout is preferable, in which each array element is preceded by its subscript number:

```
        WRITE (NPRNT,1020) NA,(I,ALIST(I),I=1,NA)
 1020   FORMAT (14H INPUT LIST OF,I5,19H  ELEMENTS, ORDERED/(I8,E16.6,
     1I8,E16.6,I8,E16.6,I8,E16.6,I8,E16.6))
```

   This gives five array elements to a line, each array element preceded by its order number (subscript) I. The format code group 5(I8,E16.6) would have been easier to write, but we are not allowed to use such a bracketed group inside another set of brackets, in Basic FORTRAN IV. It would be permissible in full FORTRAN IV.

4. The input deck structure is similar to the one outlined at the end of section A (page 130), the only exception being that we now have *two* FORTRAN decks for subprograms, one for SUBROUTINE MERGE, the other for SUBROUTINE SORT.

5. It happens frequently that some subroutines are written and should be tested, at the same time that other subroutines are not yet available. For example, at this stage, SUBROUTINE MERGE is available but SUBROUTINE SORT is not. How do we go about testing? One way, which is *not* recommended, is to alter the main

program so that it does not call upon SUBROUTINE SORT at all.

The preferred method is to produce a dummy subroutine which merely returns control, perhaps printing out that it has been entered. For example:

```
      SUBROUTINE SORT(ULIST,N)
C     DUMMY VERSION FOR TESTING PURPOSES ONLY
      RETURN
      END
```

or, more elaborately:

```
      SUBROUTINE SORT(ULIST,N)
C     DUMMY VERSION FOR TESTING PURPOSES ONLY
      NPRNT=3
      WRITE (NPRNT,1000) N
 1000 FORMAT (32H ENTERED SUBROUTINE SORT WITH N=  ,I6)
      RETURN
      END
```

All we need to do now is to supply data cards for ALIST with the values already properly ordered, and the same for BLIST. The sorting routine is then not needed, and would not have done anything significant anyway, had it been in the deck. But we do test SUBROUTINE MERGE properly by this method. This scheme of using dummy subroutines, and dummy FUNCTION subprograms which return a function value equal to zero (or equal to the argument of the function), is exceedingly useful in debugging.

Section C: COMMON Storage.

It often happens that a number of different subprograms need to use the same information. For example, the number NREAD assigned to the card reader, and the number NPRNT assigned to the line printer, are needed not only by the main program, but by other program units as well. For example, if a FUNCTION subprogram is to print out an error message when it is entered with an impossible argument value, then the FUNCTION subprogram will require access to the value of NPRNT. Putting a statement NPRNT=3 into the main program is not enough: The variable name "NPRNT" is then assigned the value three within the main program; but "NPRNT" is as undefined as ever in every subprogram (see comment number 3 on page 130).

However, FORTRAN allows us to set aside a number of memory locations as COMMON storage, meaning "common to a number of program units". For example, we may place into each program unit the FORTRAN statement

```
      COMMON NREAD,NPRNT
```

This has the effect of setting aside two memory locations in the COMMON storage area. The first of these locations contains the quantity NREAD, the second contains NPRNT. These two storage locations are "accessible" to every program unit which contains this particular COMMON declaration statement.

We still need to place some numbers into these two locations. It is advantageous to do so by means of a short, special subroutine subprogram called UNITS, which is invoked at the very beginning of the main program by the statement:

```
CALL UNITS
```

The subroutine UNITS which is invoked by this statement reads:

```
SUBROUTINE UNITS
COMMON NREAD,NPRNT
NREAD=1
NPRNT=3
RETURN
END
```

The effect of this is as follows: As soon as program execution starts, the first executable statement of the main program invokes UNITS, and subroutine UNITS in turn sets the values of NREAD and NPRNT into the first two locations of COMMON storage. (Note that UNITS must be a SUBROUTINE, not a FUNCTION. Function subprograms are not permitted to alter the values of variables in COMMON storage). Ever thereafter, these values of NREAD and NPRNT are defined and are available for use in every subprogram which contains the same COMMON statement.

We have therefore, finally and at long last, succeeded in getting around this nuisance of different unit numbers in different installations. The main program, and all the subprograms except UNITS, are now independent of unit numbers. They just use NREAD for the card reader and NPRNT for the printer. When we transfer the program from one installation to another, the only program unit which requires modification is SUBROUTINE UNITS itself.(*)

It is important to realize that a COMMON statement in a program unit gives "access", so to speak, to a certain number of memory locations in the COMMON storage area. But the *variable names used for these memory locations apply only to this one program unit. Neither the names, nor the types, nor the array dimensions, of the variables and arrays in COMMON storage need to be the same in different program units.*

This feature can be used to good advantage. For example, we may require, in the subroutine subprogram SWEAT, a "work area" of two hundred storage locations, arranged as two separate ten-by-ten matrices, both of real number type. Once used, the content of this "work area" can be forgotten. Then again, in another subroutine, say QSIZE(X), we may need a work area of 150 storage locations, as a single vector array with integer array elements. This also can be "forgotten" after use. Neither subprogram requires knowledge of NPRNT or NREAD.

It is our intention to "double up" on memory space, by using one and the same area in COMMON storage for both "work area" purposes. That is, one and the same set of memory locations will be used for the first ten-by-ten matrix in SWEAT, and for the first one hundred vector locations in QSIZE. Furthermore, the next fifty locations will be used twice also, as the first half of the second matrix for SWEAT, and as the last fifty vector locations in QSIZE.

(*) In FORTRAN II, there were special commands not requiring knowledge of any unit numbers. For example, the FORTRAN II command for output onto the line printer was: PRINT *m,List* where *m* is the statement number of a FORMAT statement, and *List* is the usual output list. Thus, in FORTRAN II, the ordinary programmer did not need to worry about silly nonsense such as what unit numbers are employed at his particular installation. The change to the "general" FORTRAN IV statement WRITE was touted as a great step forward: One statement, "WRITE", now does all the output, no matter where the output is meant to go. It should be apparent by now that this "progress" is entirely illusory; "generality", so dearly beloved by a number of systems programmers, is usually nothing but a nuisance to the working programmer who wants to get a job onto the machine, and to get answers, with minimum trouble and complications. *What is wanted is not generality, but convenience.* In our view, the vain and useless search for ever more "generality" is at the root of the present complete chaos in systems programming and job control languages.

The FORTRAN statements in SUBROUTINE SWEAT read:

```
SUBROUTINE SWEAT
DIMENSION AMATX(10,10),BMATX(10,10)
COMMON NREAD,NPRNT
COMMON AMATX,BMATX
.....
```

whereas the statements in SUBROUTINE QSIZE(X) read:

```
SUBROUTINE QSIZE(X)
DIMENSION KRAM(150)
COMMON NREAD,NPRNT
COMMON KRAM
.....
```

Explanation: Even though neither subprogram requires knowledge of NREAD or NPRNT, it would be *fatal* to omit the statement COMMON NREAD,NPRNT from either program unit. Without that statement, we would declare KRAM(1), for example, to be the first location in COMMON storage, and KRAM(2) to be the second location. If either, or both, of these array elements are altered during the course of executing the commands of the subprogram QSIZE(X), then the effect would be to overwrite, and destroy the previous contents of, these storage locations. But these very storage locations (the first two locations in common storage) are vitally important to the proper functioning of *other* program units, which use these two locations for NREAD and NPRNT!

On the other hand, as we have organized it above, the contents of KRAM can not cause this havoc. The first two locations in common storage are still reserved for NREAD and NPRNT, respectively. The array KRAM starts *after* that, i.e., KRAM(1) is now the *third* location in common storage. Similarly, AMATX(1,1) is the *third* location in common storage.

Since the contents of KRAM can be forgotten after we leave the subprogram QSIZE, and the contents of AMATX and BMATX can be forgotten after we leave the subprogram SWEAT, no difficulty arises from the fact that many of these are in fact the same storage locations in the machine. The fact that locations 3 to 152 in common storage are used as integers in QSIZE, and as real numbers in SWEAT, is also of no consequence.

Note also that it is necessary to prevent overwriting of the first two locations in the common storage area, but it is *not* necessary to assign the particular names NREAD and NPRNT to these two locations for this purpose. The statement COMMON NREAD,NPRNT could be replaced by COMMON BLANK,BLONK for example, without altering anything. All we want to do is to tell the compiler that the first two locations of common storage must not be used for KRAM, say; what these two locations are called is irrelevant.

It is apparent that common storage can be, and is, used for two quite different purposes:

1. As a *communications area* between different program units (for example, to communicate the values of NREAD and NPRNT)
2. As a *flexible work area* within any one program unit.

It is desirable to assign early locations in common storage for communications purposes, and locations farther down for work areas. Otherwise, there is too much danger of overwriting a vitally important communications area, by accident.

In full FORTRAN IV, but not in Basic FORTRAN IV, it is possible to have more than one COMMON area, with different such areas being assigned different *labels.* If this facility is available in your version of FORTRAN, by all means use it to distinguish between work areas and communication areas

Summary and Formal Definitions for Chapter XII.

1. *Rules for defining a FUNCTION Subprogram:*

   A FUNCTION subprogram is defined by a group of FORTRAN statements, the first of which is a FUNCTION statement, the last an END statement. The statements in between must contain at least one RETURN statement, and must not contain: (i) another FUNCTION statement, (ii) another END statement, or (iii) a SUBROUTINE statement. The form of the FUNCTION statement is:

   FUNCTION *FUN(AB,...,E)*

   where *FUN* is the identifier name of the function; it is related to the type (integer or real) of the function value in the usual fashion. The *dummy arguments A, B, ..., E* must be either simple variable names, or the names of arrays (in which case, a DIMENSION statement for an array by that name must appear within the FUNCTION subprogram). There must be at least one dummy argument. If there are several dummy arguments, their names must be different from each other. The actual names chosen for the dummy arguments matter only to the extent that they define the type (real or integer) of the corresponding actual arguments. (Note that dummy arguments must be identifier names; by contrast, actual arguments may also be FORTRAN constants, or may be arithmetic expressions of more general type).

   Every FUNCTION subprogram must contain at least one statement which has the effect of storing a numerical value into a location with the name of the function. The usual form of this is an arithmetic assignment statement *FUN*=... where *FUN* is the function name, and ... represents an arithmetic expression.

   Within a FUNCTION subprogram, it is strictly forbidden to alter the value of any of the dummy arguments, or of any variable in common storage. (This restriction is specific to Basic FORTRAN IV, and does not apply to full FORTRAN IV.)

   The function name must not appear in a DIMENSION statement. Neither the function name, nor any dummy argument name, may appear in a COMMON statement or in an EQUIVALENCE statement, within the FUNCTION subprogram.

   The statement RETURN, which must appear at least once and which may appear anywhere in the FUNCTION subprogram, has the effect of returning control to the superior program unit which invoked this subprogram, at the appropriate point in the superior program unit. Every possible path of control within a subprogram must lead eventually to either a RETURN statement or a STOP (or CALL EXIT) statement.

2. *Rules for defining a SUBROUTINE Subprogram:*

   A SUBROUTINE subprogram is defined by a group of FORTRAN statements, the first of which is a SUBROUTINE statement, the last an END statement. The statements in between must contain at least one RETURN statement, and must not contain: (i) another SUBROUTINE statement, (ii) another END statement, or (iii) a FUNCTION statement. The form of the SUBROUTINE statement is one of:

   SUBROUTINE *ROUT(A,B,...,E)* or SUBROUTINE *ROUT*

   where *ROUT* is the identifier name of the subroutine subprogram. In the first form above, *A, B, ..., E* are the names of dummy arguments. In the second form above, the subroutine has no dummy arguments. If there are dummy arguments, the rules of part 1 apply here also.

   The subroutine name is never used in an arithmetic assignment statement.

   Unlike a FUNCTION subprogram, a SUBROUTINE subprogram may change the values of any of its arguments and/or the values of variables in COMMON storage. Indeed, setting within the

SUBROUTINE subprogram, a numerical value for one of the dummy arguments is the normal way to supply, to the superior program unit, some number computed in the subprogram.

The name of the subroutine must not appear in a DIMENSION statement. Neither that name, nor any dummy argument name, may appear in a COMMON statement or in an EQUIVALENCE statement, within the SUBROUTINE subprogram.

The statement RETURN, which must appear at least once and which may appear anywhere in the SUBROUTINE subprogram, has the effect of returning control to the superior program unit which invoked this subprogram, at the point immediately following the CALL statement which did the invoking. Every possible path of control within the subprogram must lead eventually to either a RETURN statement or a STOP (or CALL EXIT) statement.

3. *Rules for Actual Arguments:*

   1) If the dummy argument *A* is the name of a simple (non-array) variable which is not altered within the subprogram, then the corresponding actual argument *AP* may be any FORTRAN constant, variable name, or well-formed arithmetic expression, of the same type (integer or real) as the type of *A*.

   2) If the dummy argument *A* is the name of a simple (non-array) variable which *is* altered within the subprogram (this can happen only in a SUBROUTINE subprogram), then the corresponding actual argument *AP* must be a variable name of the same type as *A* (that is, constants, or more complicated arithmetic expressions, are then not allowed).

   3) If the dummy argument *A* has been declared within the subprogram as the name of an array (by means of a DIMENSION statement), then the corresponding actual argument *AP* must also be the name of an array, of the same type and of the same precise dimensions. In practice, however, many FORTRAN compilers ignore the rule about "same precise dimensions" for *vector* arrays; the rule is enforced strictly for *matrix* arrays.

4. *Rules for Invoking Subprograms:*

   In order to invoke a FUNCTION subprogram FUNCTION *FUN(A,B,...,E)* the superior program unit has to contain an arithmetic expression involving the term *FUN(AP,BP,...EP)* where *AP* is an appropriate actual argument for *A*, *BP* is an appropriate actual argument for *B*, and so on. The value of *FUN* is then computed with the numerical values of the actual arguments, and this function value is returned to the superior program unit, which then proceeds with the remainder of the evaluation of the arithmetic expression.

   In order to invoke a subroutine subprogram SUBROUTINE *ROUT* without dummy arguments, the superior program unit must contain the statement

   CALL *ROUT*

   If the subroutine subprogram has dummy arguments, e.g., SUBROUTINE *ROUT(A,B,...,E)*, then the superior program unit must contain the statement

   CALL *ROUT(AP,BP,...,EP)*

   where *AP* is an appropriate actual argument for *A*, and so on. The effect of the CALL statement is to transfer control to the subroutine subprogram. The commands of the subprogram are obeyed, with actual arguments replacing the dummy arguments. When a RETURN statement is reached, control reverts to the superior program unit, at the executable statement immediately following the CALL statement.

5. *Rules for the COMMON Statement:*

   The COMMON statement has the form

   COMMON *A,B,...,E*

   where *A*, *B*, ..., *E* are either variable names or array names. If several COMMON statements occur in one program unit, the lists are strung together one after the other, as if there had been a single COMMON statement with the combined list. For example, the statements

```
COMMON JOY,HOPE
COMMON BLACK,DEATH
```

are equivalent to the single statement

```
COMMON JOY,HOPE,BLACK,DEATH
```

The effect of the COMMON statement is to reserve a number of storage locations in COMMON storage, one storage location for each simple (non-array) variable name, and the appropriate number of storage locations for each array name. For example, if JOY, HOPE, and BLACK are simple variable names, and DEATH is the name of a vector array of dimension thirteen, then the COMMON statement above reserves sixteen storage locations in COMMON storage. The first of these locations is called JOY within this program unit, the fourth location is called DEATH(1), and so on. These names are "local" to this particular program unit, however. Other program units may reserve different amounts of common storage, and may refer to any or all of these storage locations by different names.

Two quantities, both of which appear in COMMON statements within one program unit, must not be declared equivalent to each other by an EQUIVALENCE statement. However, EQUIVALENCE statements may be used to declare a quantity in COMMON, equivalent to some other quantity not appearing in a COMMON statement. The effect is to place the second quantity into COMMON storage, also. For example, the three statements:

```
DIMENSION JUNK(300)
COMMON A,B
EQUIVALENCE (B,JUNK(1))
```

have this result: A COMMON storage area of length 301 storage locations is set aside. Within this program unit, the first location in that area is called A. The second location has two names: B and JUNK(1). The third location is called JUNK(2); and so on till the 301st location, which is called JUNK(300). Note that the EQUIVALENCE statement has had the effect of lengthening the area set aside for COMMON storage, from the original two locations for A and B, to 301 locations. This process is called "implicit extension of COMMON storage".

The only lengthening of COMMON storage permitted is a "forward" lengthening, as in our example. "Backward" extension is forbidden. For example, the following statements are *invalid* FORTRAN:

```
DIMENSION JUNK(300)
COMMON A,B
EQUIVALENCE (B,JUNK(5))
```

This is *invalid* because it amounts to an attempt to lengthen COMMON in a "backward" direction. B is declared synonymous with JUNK(5), thereby making A synonymous with JUNK(4). The first three locations of the array JUNK would then have to be (i) in COMMON, and (ii) ahead of A. This is inconsistent with the COMMON statement, which declares that A is the name of the *first* location in COMMON storage.

6. *Sequence of Statements within a Program Unit:*

In Basic FORTRAN IV, certain types of FORTRAN statements must appear in a definite order. Before giving this order, we define "executable" and "non-executable" statements as follows: The non-executable statements are: FUNCTION, SUBROUTINE, DIMENSION, COMMON, EQUIVALENCE, Arithmetic Function Definitions, FORMAT statements, and the END statement. All other statements are executable.

The sequence of statements is:

1. FUNCTION or SUBROUTINE statement (in a subprogram); nothing for the main program.
2. All DIMENSION statements.
3. All COMMON statements.
4. All EQUIVALENCE statements.
5. All Arithmetic Function Definitions.
6. Executable Statements and FORMAT Statements, in any sequence.
7. The END statement.

Comment cards are not counted as FORTRAN statements, and may be put anywhere. The strict rules for the sequence of statements are relaxed somewhat in full FORTRAN IV, and are relaxed even more in some versions of FORTRAN. But we recommend adhering to these rules nonetheless.

DRILL EXERCISES FOR CHAPTER XII.

1. What is the first statement in a FUNCTION subprogram? What is the purpose of the RETURN statement? How is a value assigned to the function? What determines the type (real or integer) of the function value?

2. The function ROOT(X,N) appearing on page 125 is supplied along with the following main program. State what will be printed out.

```
      NPRNT=3
      U=3.0
      Y=5.0*ROOT(U**4-17.0,3)+2.5
      WRITE (NPRNT,1200) Y
1200 FORMAT (3H Y=,F8.2)
      STOP
      END
```

3. The function INDEX(X,OLIST,N) appearing on pages 127-128 is supplied along with the following main program:

```
      DIMENSION WHAT(22)
      NREAD=1
      NPRNT=3
      READ (NREAD,1000) M,(WHAT(I),I=1,M)
1000 FORMAT (I5/(4E10.0))
1500 READ (NREAD,1550) WHY
1550 FORMAT (E10.0)
      IF(WHY) 2000,2000,1580
1580 JACK=INDEX(WHY,WHAT,M)
      WRITE (NPRNT,1600) WHY,JACK
1600 FORMAT (E16.6,18H  FITS IN POSITION,I6)
      GO TO 1500
2000 STOP
      END
```

The following data cards are supplied, with the numbers in appropriate fields:

```
Data card 1:    3
Data card 2:    2.5      10.6     35.9
Data card 3:    1.7
Data card 4:   98.4
Data card 5:    3.8
Data card 6:    blank
```

State what is printed out by this program.

4. Draw flow charts and write FUNCTION subprograms for the following functions:
   a) ZIGGY(X) is equal to the absolute value of X if X lies in the range -1.0 to +1.0. For other values of X, ZIGGY(X) is defined by the condition that it is a periodic function of X with period 2; that is, for all values of X, ZIGGY(X) is equal to ZIGGY(X-2.0) and to ZIGGY(X+2.0).
   b) SQWVE(X) is zero if X is numerically equal to an integer (for example, if X equals 3.0); the function value is +1.0 if X lies between 2*N and 2*N+1, where N is any integer, and the value is -1.0 if X lies between 2*M+1 and 2*M+2, where M is some integer. (This is the "square wave" function used extensively in electrical engineering).
   c) SMALL(ARRAY,N) returns the smallest value among the first N values stored in the vector ARRAY. Assume that ARRAY has dimension 100 in the calling program.
   d) TRACE(ATRIX) returns the sum of the diagonal values of the ten-by-ten matrix array ATRIX. (This sum is called the "trace" of the matrix.)

5. Given the following main program and SUBROUTINE subprogram, what numerical values are printed out in execution?

```
C     MAIN PROGRAM
      NPRNT=3
      I=1
      J=2
```

```
      K=3
      CALL MADD(I,J,K,MOAN)
      WRITE (NPRNT,1300) MOAN,I,J,K
1300  FORMAT (4I12)
      STOP
      END
      SUBROUTINE MADD(L,M,N,MOO)
      MOO=L+M+N
      L=L+1
      M=M+2
      N=N+13
      RETURN
      END
```

6. A matrix of size N-by-N, stored in a matrix array of dimension ten-by-ten, is to be *transposed.* That is, the matrix elements ATRIX(I,J) and ATRIX(J,I) are to be interchanged, for all I and J between 1 and N. Write a SUBROUTINE TRNSP(ATRIX,N) to achieve this.

7. The SUBROUTINE PUTIN(X,K,VEC,N) inserts the argument value X into the K'th element of the vector array VEC, which has N elements used (assume N is greater than or equal to K-1). The dimension assigned to VEC in the calling program is 100. The K'th original vector element is pushed forward into position VEC(K+1), the (K+1)th element is pushed into position VEC(K+2), and so on. Also, the argument value N is increased by one unit.

8. The SUBROUTINE REMOV(K,VEC,N) removes the K'th element of the vector array VEC, which has N elements used (assume N is greater than or equal to K). The dimension of VEC in the calling program is 100. VEC(K+1) is "pushed down" into the VEC(K) location, then VEC(K+2) is pushed down into the VEC(K+1) location, and so on (can you explain why the order in which this is done matters greatly?). VEC(N) itself is cleared to zero, and N is altered to N-1.

9. Assume the following main program and subroutine subprogram are executed:

```
      COMMON X,Y,Z
      X=2.0
      Y=3.0
      CALL RAT
1500  X=Y*Z+X
      CALL RAT
1700  T=X*Z
      STOP
      END
      SUBROUTINE RAT
      COMMON P,Q,R
      R=P+Q
      P=Q*R
      RETURN
      END
```

   a) What are the contents of the first three locations in COMMON storage when SUBROUTINE RAT is first invoked?
   b) when the RETURN statement is reached in RAT?
   c) after statement 1500 of the main program has been executed?
   d) What value is finally stored into location T?

10. How many storage locations are reserved in COMMON storage by the following statements:
   a) COMMON A,B,Q,K,L
   b) DIMENSION ARTY(12,14)
      COMMON ARTY,FANCY
   c) DIMENSION HAIRY(12,10),FACE(120)
      COMMON JUNK
      EQUIVALENCE (HAIRY(1,1),FACE(1),JUNK)
   d) first two statements same as in (c), but the third statement now reads:
      EQUIVALENCE (HAIRY(1,1),FACE(60)), (FACE(1),JUNK)

PROGRAMMING EXERCISES FOR CHAPTER XII.

1. Write and test a SUBROUTINE SORT(ULIST,N) which sorts the first N elements of the vector array ULIST in increasing order. Note that "testing" involves writing a main program which reads in values of N and ULIST, calls upon SORT, then prints out results.

2. Write and test a function evaluation program for the "inverse hyperbolic tangent" function ATANH(X). The mathematical formula for this function reads:

   $\text{arc tanh } x = \frac{1}{2} \log_e[(1+x)/(1-x)]$

   where the logarithm can be generated from the built-in function ALOG. In this form, the function could be defined by means of an arithmetic statement function definition, see Chapter IX. However, this mathematical formula becomes inaccurate for purposes of computation, and should therefore not be used, when the absolute value of x is small, say below 0.01. For such values of x, it is better to use the first five terms of the series:

   $\text{arc tanh } x = x + x^3/3 + x^5/5 + x^7/7 + x^9/9 + \ldots$

   (Note that the need to choose between two different methods of evaluation of the function, depending upon the actual value of the argument x, makes it impossible to write this as an arithmetic statement function!)
   The function is *undefined* if the absolute value of x is larger than 1. Error messages should be printed out for such arguments, and the function value should be set to zero.
   To test this function evaluation routine, the main program reads numbers x from data cards, evaluates y=tanh(x) by using the built-in function TANH, and then evaluates z=arctanh(y) by invoking the function subprogram; we print out x, y, and z. x and z should be equal.

3. Write and test a set of matrix operation subroutines, called MADD, MSUB, and MMPY, respectively, for adding, subtracting, and multiplying two N-by-N matrices A and B, storing the resulting matrix into matrix array C. A, B, and C are assumed to be dimensioned as ten-by-ten matrix arrays. If N equals 10, the full arrays are used; if N is less than 10, the arrays are used only partially, with a number of array elements "unemployed" in the machine.
   Matrix addition and subtraction are defined as addition and subtraction of corresponding elements in the two matrices. Matrix multiplication of two N-by-N matrices is defined by:

   C(I,J) = A(I,1)*B(1,J) + A(I,2)*B(2,J) + A(I,3)*B(3,J) + ... + A(I,N)*B(N,J)

   Note that the array names A, B, C, and the value of N, must be dummy arguments.

4. (More advanced exercise) Rewrite the matrix routines of exercise 3, assuming that all the matrix elements are stored "tightly" within vector arrays. For example, the N-by-N matrix A is stored into the first N**2 elements of the vector array AVEC, in such a way that the matrix element A(I,J) is stored into AVEC(K) where K=I+(J-1)*N. In this way, no storage locations are "unemployed". The dummy arguments of the subroutines are AVEC, BVEC, CVEC, and N, in that order.

5. (Even more advanced exercise) In many practical applications of matrices in mathematical programming, the matrices are symmetric, i.e., the matrix elements A(I,J) and A(J,I) are always equal to each other. The method of storage into vector arrays described in exercise 4 is then wasteful of storage space. For example, a general ten-by-ten matrix has 100 different matrix elements, but if the matrix is symmetric, only 55 matrix elements are really needed. The other 45 elements (say, the elements below the main diagonal of the matrix) can be deduced by using the symmetry property of the matrix. Devise a method for "tight packing" of the elements of a symmetric matrix of size N-by-N into a vector array with (N*(N+1))/2 storage locations, and alter the subroutines of exercise 4 to allow for this new method of packing.

6. Using your present knowledge of subprograms and of built-in functions, write a FUNCTION subprogram IDIV(K,L) which produces, as the function value, the properly rounded integer value nearest to the true quotient K/L. Make sure to allow for all possibilities, i.e., each of K or L may be positive, negative, or zero. If L is zero, print out an error message and stop execution.

7. Set up a program for the following, highly oversimplified, accounting system (our abject apologies to the accounting profession, but the programming principles involved are not altered at all when more realistic accounting procedures are employed).

Input cards: One card for each "business event", in FORMAT (4I5,3F10.2); the first four integers are, in that order, the event type number (see below), the number of the department (there exist five departments in this company), the number of the product line (item) involved (each department handles ten separate product lines), and the number of items purchased or sold or returned, as the case may be (any one event involves at least one item, and never more than 5000 items). The three remaining fields contain money values in fractional form, for example, "43.65" for 43 dollars and 65 cents. The event types and their meanings are:

Event type:	*Meaning of event:*	*Meaning of final three fields:* *First field:*	*Second field:*	*Third field:*
1	Sale for cash	Sales price (per item)	Percent sales tax applicable	Inventory value (per item)
2	Purchase of stock to add to inventory	Purchase price (per item)	blank	blank
3	Return of sold merchandise	Sales price (per item)	Percent sales tax applicable	Inventory value (per item)
4	Write-off of spoiled items in inventory	blank	blank	Inventory value (per item)

For example, if product line 7 of department 2 (a gold-plated motor-driven eggbeater) is attacked by clothes moths which eat up the insulation in the motor, and 2500 of these eggbeaters have to be written off completely, then the input card for this sad event contains the numbers, in sequence: 4, 2, 7, 2500, blank, blank, 399.98 (where the last number is the inventory value of one such eggbeater).

Accounts kept: Each account is stored as a five-by-ten matrix array with integer elements representing numbers of cents. For example, the matrix array NCASH, with 5 rows and 10 columns, contains cash amounts in the various departments, associated with the various product lines; NCASH(2,7) is the cash amount (in cents) held in department 2 in relation to product line 7. The following accounts are maintained:

NCASH Cash amount (asset)
NVENT Value of inventory (asset)
NTAX Sales tax payable (liability)
NSALE Gross value of sales, before tax (revenue)
NPUR Value of purchases to replenish inventory (costs and expenses)

Programming considerations: The main program is used only to organize the overall work. The details are all done by subroutine subprograms. All accounts, as well as the quantities read in from the latest input card, are kept in COMMON storage. SUBROUTINE READR(KO) reads one input card at a time, echo-checks it, stores numbers into appropriate locations in COMMON; it tests the numbers, returning KO=1 if the card was obviously wrong (e.g., a department number not in the range 1 to 5), KO=2 otherwise. In the latter case, a card image of the faulty card is also printed out. For correct cards, the main program tests the value of the event type number (with a computed GO TO, see page 83), and calls one of the four subroutines SELL, BUY, SAD, and AWFUL, respectively. Each subroutine does whatever additional checking is appropriate (for example, sales tax rates are between one and five percent on all items, and nothing is ever sold at a loss). If these tests fail, an error message is produced, and no calculation is done. If the tests succeed, the "affected" accounts are updated.

8. Modify exercise 7, by including "event type 5", which is a request for a *management flash report.* This contains, *in easily readable form,* the cash position, inventory position, total assets, sales tax liability, accumulated sales, and accumulated purchases for each of the five departments as a whole.

9. Augment the flash report by printing out the top two, and the bottom two, product lines as measured by the ratio of (net value of sales)/(accumulated value of purchases). Produce full identification: department number, product line number, money values, value of ratio.

10. To guard against machine failure, we count the business event cards. Every 100 cards, we call GUARD, which prints out, and punches out in card form, all numbers in all accounts. Business "event type 6" is a request for a *restart,* using these cards as input.

CHAPTER XIII

PROGRAM PLANNING AND DEBUGGING

Section A: General Principles.

Only a beginner imagines that writing down a FORTRAN code is all that there is to computer programming. In fact, this is only a small fraction of the total time and effort involved in getting a computer program "on the air". The major effort must go into *program planning* and into *debugging*.

Some of the questions which should be asked during *program planning* are:

1. What method of calculation should be used? What are the storage space requirements of the available methods, and what are the estimated machine times needed?
2. Will we need to employ auxiliary storage, such as tapes and disks? If so, which of these should be used? How do we protect ourselves against machine failure and/or operator error during execution of the job? How do we protect ourselves against accidental loss or destruction of a tape, or of a disk?
3. How should the program be broken up into program units? What information should be placed into COMMON storage to ensure easy communication between subprograms? How much COMMON storage should be set aside for work areas?
4. What data shall we need to read in from data cards? What checks can we make to protect ourselves against incorrectly punched data, or data cards out of order?
5. Are we likely to encounter numerical troubles arising from loss of precision during the course of the computation (see Chapter VI)? If so, how can we detect this when it happens, and what can we do about it?
6. What output do we want? In what form, or forms, do we want this output (printed paper, punched cards, output magnetic tape, ...)?
7. At what points in the program do we want debugging output, and what debugging output do we want at each such checkpoint? How do we check the flow of control during program execution? What data numbers will be needed to test all eventualities that we can think of?

All these questions, and more, are asked by the experienced programmer. All of them should be answered clearly and precisely, *before* any FORTRAN coding or even flow charting is done at all. Important as all these questions are, by far the most important of them is question seven.

All programs are crawling with bugs. The debugging operation has to be planned as an integral part of planning the program itself. Of course, the simple errors of FORTRAN grammar are detected, and flagged, by the FORTRAN compiler. But after those minor bugs are removed, and the program compiles without diagnostic messages, then we are faced with the really vicious bugs.

Remember, the machine obeys your instructions literally, without thinking. It does what you *tell* it to do, not what you *want* it do do! If there are no compiler diagnostics, then your instructions are written in grammatically correct FORTRAN. But it is entirely possible to say utter nonsense in grammatically correct English, and also in grammatically correct FORTRAN.

Debugging is an art, not a science. All we can do is give a few hints and suggestions, based on experience. But each programmer has to learn to recognize those errors to which he himself is particularly prone, and he has to train himself to plan his debugging to catch these particular types of error (as well as all other types). The following points are worth remembering:

1. Write *clean code,* not tricky, fancy coding. It is surprising how much easier it is to debug a straightforward program, than one in which one has tried to be "clever". Draw up a simple, clean flow chart, and follow it. Insert *plenty of comment cards* at the beginning of each program unit, as well as interspersed throughout the code itself. These comments will be worth their weight in gold later on.

2. Make sure to *echo-check* every data number read into the machine, immediately after reading it. Without echo-check printout, debugging is hopeless. If you don't know what is in the machine initially, how are you going to tell what happened afterwards to produce the crazy nonsense printed out by the machine? Looking at the data cards tells you what you *meant* to read into the machine. It does *not* tell you what the machine itself stored away in its memory. The echo-checking must be done immediately, for two reasons:
   (i) The program might "blow up" before you ever get to a delayed echo-check. Then you don't know what numbers caused the blow-up.
   (ii) The contents of storage locations, into which input numbers have been read, can alter during program execution, either deliberately or as the result of a bug. If they alter before the (delayed) echo-check, the echo-check becomes positively misleading since it fails to reproduce the original data numbers.

3. There must be extensive *debugging output* during program execution. In the early stages of debugging, it is advisable to test the "flow" of program execution, that is, to print out that we have reached a certain checkpoint in the program, whenever we reach that point, and to do so for a number of separate, suitably chosen checkpoints. Later on, after the most glaring errors of flow have been corrected, the time has come to check the actual numbers produced by the program, at various intermediate stages of program execution. Most of these numbers are not final answers worth printing out in a production run. But unless intermediate numbers are printed out, there is no practical way of telling why the final answers are such complete nonsense. Later on, in section B, we shall present methods for flow testing and for producing debugging printout.

4. The final essential for debugging is *full recomputation of several typical cases,* on an electric desk machine. All numbers printed out by the computer, intermediate results as well as final results, must be checked by hand, in full detail. Careful planning is required for choosing the "typical cases". We must pick simple cases for recomputation, otherwise we do not live long enough to do the checking. But these cases must not be so simple, or so short, that they miss out on essential steps which happen during normal production running of the program. The choice of what to recompute, like all of the delicate task of debugging, is an art, not an exact science.

You may ask, why use an electronic computer at all, if we have to check the beast every inch of the way? Well, once a program has been debugged fully, this program can be used many times over, with different sets of input data. If the program is a standard subroutine, then this subroutine can be used (after debugging) as a program unit in a large number of different programs. In the long run, we do win. But in the short run, it is pure hell.

After all tests have succeeded, after all numbers have been checked and verified by hand, after everything known to the wit of man has been done to remove the bugs from a program, then and only then is it no longer quite certain that the program is full of bugs. Then it is merely very likely.

Section B: Some Useful Techniques.

In the remainder of this Chapter, we present particular techniques for flow testing and for debugging printout (also called "dynamic dumping"). We have used these techniques and found them effective.

For flow testing, we write a subroutine subprogram FLOW with two arguments, M and N. The idea is that M is the number of the program unit from which flow is being invoked (say, M is 1 for the main program, M is 2 for one of the subprograms, M is 3 for another subprogram, and so on), and N is chosen to be the statement number of a nearby statement in that program unit, near to the point at which we invoke FLOW. For example, we may wish to insert a flow checkpoint into program unit "3", near statement number 1500 in that program unit. To achieve this, we insert, into that program unit, the statement:

```
CALL FLOW(3,1500)
```

The effect of this will be to print out a message whenever program execution passes through this particular checkpoint; the message in this case reads:

```
FLOW CHECK. WE ARE IN PROGRAM UNIT  3     NEAR STATEMENT NUMBER     1500
```

The FORTRAN code for SUBROUTINE FLOW(M,N) follows. Note that we have assumed that the first two locations of COMMON storage contain the unit numbers NREAD and NPRNT, respectively.

```
      SUBROUTINE FLOW(M,N)
      COMMON NREAD,NPRNT
      WRITE (NPRNT,1000) M,N
 1000 FORMAT (35H FLOW CHECK. WE ARE IN PROGRAM UNIT,I3,4X,21HNEAR STATE
     1MENT NUMBER,I8)
      RETURN
      END
```

Next, we turn our attention to *dynamic dumping,* i.e., to the printout of intermediate numbers during program execution. It is desirable to arrange it in such a way that we can alter the checkpoints from which we want such printout, merely by supplying a different data card (rather than by recompiling the FORTRAN code). It is also desirable to limit the number of times that we get printout from any one checkpoint in the program. Dynamic printout the first few times is most valuable, but after that, nothing new is likely to emerge from studying more and more printouts from the same checkpoint. On the other hand, cases do arise when we want a large number of printouts from one checkpoint, for example, when we know (from other checkpoint printout) that trouble arises during the execution of a particular loop, but we do not know just when in the loop the trouble occurs (for example, the trouble may be an arithmetic overflow). Thus, we want a limit, but a flexible limit, on the number of printouts from any one checkpoint.

The following technique achieves these objectives. We start by defining a vector storage area in COMMON storage, to serve as a communications area for debugging control. Let us call this vector NOBUG and let us give it a dimension of twenty, say, for a small program (much more than that for an enormous program). Each array element of the array NOBUG will control one checkpoint.

For example, the value of NOBUG(5) decides whether, and how much, printout we get from checkpoint number five. The convention we use is:

i) NOBUG(5) zero means no printout from checkpoint five;
ii) NOBUG(5) negative means unlimited printout from checkpoint five;
iii) NOBUG(5) positive, say equal to 9, means that number (9) of printouts from checkpoint five. The tenth time we reach this checkpoint, there is no more printout.

The values of all the array elements of array NOBUG are read in at the very start of the execution phase, by SUBROUTINE UNBUG. Different data for NOBUG lead to different debugging printouts from the same program. (Naturally, it is desirable to maintain a list of the various checkpoints which have been built into the program, with a short indication of what kind of printout occurs at each checkpoint; comment cards at the beginning of the main program are well suited for this).

Let us now show the start of the main program, the coding for SUBROUTINE UNBUG, and a typical debugging sequence using the value of NOBUG(5), within one of the subprograms.

```
C       MAIN PROGRAM
C       ... (LOTS AND LOTS OF COMMENTS)
        DIMENSION NOBUG(20)
        ... (OTHER DIMENSION STATEMENTS)
        COMMON NREAD,NPRNT,NOBUG
        ... (OTHER COMMON STATEMENTS)
        ... (EQUIVALENCE STATEMENTS, IF ANY)
        ... (ARITHMETIC FUNCTION DEFINITIONS, IF ANY)
        CALL UNITS
C       THIS SETS THE VALUES OF NREAD AND NPRNT
        CALL UNBUG
C       THIS READS IN VALUES FOR NOBUG, FROM DATA CARD
        ... (ACTUAL COMPUTATION STARTS HERE)
```

The subroutine subprogram UNITS has been given on page 136. The new subprogram is UNBUG:

```
        SUBROUTINE UNBUG
        DIMENSION NOBUG(20)
        COMMON NREAD,NPRNT,NOBUG
        READ (NREAD,1000) NOBUG
 1000 FORMAT (20I3)
        WRITE (NPRNT,1200) NOBUG
 1200 FORMAT (16H DEBUG CONTROL =,20I5)
        RETURN
        END
```

Now let us suppose that checkpoint number five, the one controlled by the value of NOBUG(5), sits within SUBROUTINE HEAVE(HO), between statements number 1600 and 1700 of that subprogram. If printout from this checkpoint is called for, then we want to see the name of the subroutine, the place we are within that subroutine (i.e., the statement number "1600"), the value of the dummy argument HO, as well as values of three integer variables, say I, J, and K. To do this, we place the following statements within SUBROUTINE HEAVE(HO):

```
        SUBROUTINE HEAVE(HO)
        DIMENSION NOBUG(20)
        COMMON NREAD,NPRNT,NOBUG
        ...
 1600 ... (WHATEVER IT IS)
C**** DEBUGGING
 1601 IF(NOBUG(5)) 1603,1608,1602
 1602 NOBUG(5)=NOBUG(5)-1
 1603 WRITE (NPRNT,1604) HO,I,J,K
 1604 FORMAT (22H HEAVE,1600. HO,I,J,K=,E16.6,3I12)
 1608 CONTINUE
C**** END DEBUGGING
 1700 ... (WHATEVER IT IS)
```

Note that statement 1602 is reached if and only if NOBUG(5) is positive. The effect of statement 1602 is to decrease the value of NOBUG(5) by one unit. If NOBUG(5) was equal to nine, at the beginning, then it is decreased to eight the first time we reach this checkpoint, to seven the next time we reach that checkpoint, and so on. The ninth time we reach this checkpoint, the value of NOBUG(5) is decreased from one to zero, and ever thereafter, this checkpoint produces no more printout, and does not change the value of NOBUG(5) any further: from now on, we skip from statement 1601 straight down to statement 1608. The CONTINUE statement, number 1608, does not terminate any DO loop; rather, it serves to make this debugging section of subroutine HEAVE(HO) into a self-contained unit. This unit can be removed in its entirety (including the comment cards at the beginning and at the end) without affecting the rest of the program unit in any way.

The output statement 1603 is reached if NOBUG(5) differs from zero. The FORMAT statement produces a message in front of the numbers, to tell us that we are in subprogram HEAVE, near statement number 1600 of that subprogram, and that the numbers which follow are the values of HO, I, J, and K, respectively. Only then come the numbers!

It takes time and effort to think up what checkpoints are required, and what output we want at each checkpoint. It takes time and effort to program in all these checkpoints. But for every minute spent on this, you will save hours of misery trying to figure out, from entirely inadequate information, just how the program went on the rocks so horribly.

A number of manufacturers provide "Debug Packages" for obtaining dynamic dumps. We recommend *against* use of these facilities. Standard FORTRAN IV contains all that is needed. It is dangerous to become dependent upon the special facilities provided by one manufacturer, on one machine. By the time one learns how to use the debug package properly, two things are likely to happen: (1) All the bugs could have been found in the same time, using the FORTRAN techniques given above; and (2) The system provided by the manufacturer changes, or the machine is replaced by a newer machine, so that one has to learn an entirely new debug package!

Of course, all these objections will be overcome, null and void, if and when debug facilities are included as part of USA Standard FORTRAN IV. Devoutly as one might pray for that happy day, it has not arrived yet.

A GLOSSARY OF BASIC FORTRAN IV

1. *Character Set:*

   Numeric: 0 1 2 3 4 5 6 7 8 9

   Alphabetic: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   Special: Blank, equal "=", plus "+", minus "-", asterisk "*", slash "/", left parenthesis "(", right parenthesis ")", comma ",", period "."

   Alphameric: Means either alphabetic or numeric, not special.

2. *Card Layout:*

   Comment Card: C in column 1, anything thereafter; not translated into machine language.
   END Card: END in columns 7, 8, 9. Must be the last card of every program unit.
   Statement Cards: *Initial card* may have a statement number (between one and four digits, no sign) in columns 1 to 5, must have a blank or "0" in column 6, the statement itself in columns 7 to 72, inclusive. *Continuation cards* have no statement number, must have a character (other than "0") in column 6, and continuation of the statement in columns 7 to 72. No more than five continuation cards are allowed.

3. *Constants:*

   Integer constant: signed or unsigned string of numeric characters, no decimal point; must not exceed maximum permissible value for the particular machine.
   Real constant: (basic) signed or unsigned string of numeric characters, with a decimal point included; (E-form) basic real constant, followed by letter "E", followed by an integer, e.g., 4.57E-13.

4. *Identifier Names:*

   Between one and five alphameric characters, the first of which must be alphabetic. A first character I, J, K, L, M, or N indicates integer values, the others real values.

5. *Arrays:*

   Array names declared as such in DIMENSION statements; vector arrays (one subscript) and matrix arrays (two subscripts); for permissible subscript expressions, see page 100; for order of array elements of a matrix array in storage, see page 108.

6. *Lists:*

   Simple list: Identifier names, separated by commas; no comma at beginning or end of list; the list may contain names of simple variables, array names, and array element names.
   Implied DO loop list item: Used only in input/output lists; see page 117.

7. *Arithmetic Operators:*

   "+" "-" "*" "/" and "**" (raising to a power)
   Mixed mode expressions are forbidden (see page 17). Hierarchy of operations on page 18.

8. *Types of Statements:*

   Executable: Assignment statements, control statements, input/output statements.
   Non-executable: FUNCTION and SUBROUTINE statements, specification statements, arithmetic function definition statements, FORMAT statements, END statement.

9. *Program, and Program Units:*

   Program consists of program units, one of which is the main program; all program units terminate with an END card; for rules on FUNCTION and SUBROUTINE subprograms, see pages 138-140.

10. Reference Table of Statement Types:

Type and Subtype:	*Statement:*	*Typical Example:*	*Page:*
EXECUTABLE:			
Assignment:	Assignment	X=Y**2+3.0*(Z+A/Y)	18
Control:	GO TO	GO TO 2300	33
	computed GO TO	GO TO (2300,2400,2500),INDEX	92
	IF	IF(N-3) 2300,1200,4000	33
	STOP	STOP	33
		CALL EXIT (in some places)	33
	PAUSE	PAUSE 7777	92
	DO	DO 2300 JOE=3,25,2	92
	CONTINUE 2300	CONTINUE	92
	CALL	CALL HEAVE(HO)	139
	RETURN	RETURN	138
Input/ouput:	WRITE	WRITE (NPRNT,2500) A,B,VECTR	44,79
		WRITE (NTAPE) NREC,ARRAY	123
	READ	READ (NREAD,2800) C,D	44,79
		READ (NTAPE) NREC,ARRAY	123
	REWIND	REWIND NTAPE	123
	BACKSPACE	BACKSPACE NTAPE	123
	END FILE	END FILE NTAPE	123
NON-EXECUTABLE:			
Subprogram Header:	FUNCTION	FUNCTION INDEX(X,OLIST,N)	138
	SUBROUTINE	SUBROUTINE HEAVE(HO)	138
Specification:	DIMENSION	DIMENSION VEC(25),ARR(8,12)	112
	COMMON	COMMON NREAD,NPRNT,NOBUG	139
	EQUIVALENCE	EQUIVALENCE (A,VEC(3))	112
Statement Func'n Def.		TAN(A)=SIN(A)/COS(A)	98
FORMAT:	FORMAT 2500	FORMAT (3H X=,F12.5//)	79
	Field descriptors:	*n*F*w.d*, *n*E*w.d*, *n*I*w*, *n*X, *n*H...	79
	Field separators:	Comma, Slash (end of record)	79
END	END	END	11

11. Sequence of Statements within one Program Unit:

1. FUNCTION or SUBROUTINE statement (if any);
2. DIMENSION statements;
3. COMMON statements;
4. EQUIVALENCE statements;
5. Statement Function Definitions;
6. Executable Statements and FORMAT Statements, in any sequence;
7. END Statement.

Note: Comment cards can be anywhere, are not counted as statements.

12. Differences between USA Standard Basic FORTRAN IV and USA Standard FORTRAN IV:

(*Note:* Page references prefaced by "IFP" refer to "Introduction to FORTRAN IV Programming" by J.M. Blatt, Goodyear Publishing Co., California, USA, 1968).

1. Character set: extra (special) symbol "$"
2. Card Layout: Up to nineteen continuation cards permitted
3. Constants: Double precision and complex constants (IFP61, IFP62), logical constants (IFP83)
4. Identifier names: Up to six characters permitted; explicit type declarations exist (IFP72)
5. Arrays: Three-subscript arrays permitted; arrays can be declared in COMMON and/or explicit type declaration statements (IFP112, IFP179)
6. Lists: No change
7. Operators: No change in arithmetic operators; but logical operators exist (IFP18)
8. Types of statements: *New types are:* Logical assignment (IFP83), Assigned GO TO and ASSIGN (IFP89), Logical IF (IFP21), BLOCK DATA subprogram header (IFP183), labelled COMMON (IFP180), EXTERNAL (IFP155), adjustable dimensions (IFP162), explicit type declarations INTEGER, REAL, DOUBLE PRECISION, and LOGICAL (IFP72), DATA declaration (IFP69), plus the following extensions to the FORMAT statement: second-level bracketed groups permitted (IFP207), field descriptors *n*D*w.d* (IFP196), *n*G*w.d* (IFP198), *n*L*w* (IFP210), *n*A*w* (IFP210,IFP220-225), scale factor option "P" (IFP197), reading and writing character strings (including format information) into and out of core storage (IFP220-228), printer control (IFP200).
9. Program Units: BLOCK DATA subprogram (IFP183); FUNCTION allowed to alter its dummy arguments and/or quantities in COMMON storage.
10. See list under item 8.
11. Sequence of Statements: DIMENSION, COMMON, and EQUIVALENCE may come in any order, but must still precede all arithmetic function definitions.

13. Built-in Functions in Basic FORTRAN IV and in FORTRAN IV:

Basic FORTRAN IV (see page 95): ABS, IABS, ALOG, EXP, SIN, COS, SQRT, TANH, ATAN, IFIX, SIGN, ISIGN (the last two are functions of two variables)

FORTRAN IV (IFP103-106): Starting letter "D" for "double precision", "C" for "complex";

the additional functions of *one* variable are: DABS, CABS, AINT, INT, IDINT, SNGL, REAL, AIMAG, DBLE, CONJG, DEXP, CEXP, DLOG, CLOG, ALOG10, DLOG10, DSIN, CSIN, DCOS, CCOS, DSQRT, CSQRT, DATAN

the additional functions of *two* variables are: AMOD, MOD, DMOD, DSIGN, DIM, IDIM, CMPLX, ATAN2, DATAN2

the additional functions of *two or more* variables are: AMAX0, AMAX1, MAX0, MAX1, DMAX1, AMIN0, AMIN1, MIN0, MIN1, DMIN1

SOLUTIONS TO DRILL EXERCISES

In this section, we give answers to the drill exercises. However, in many cases these answers are not unique. If the drill exercise involves writing FORTRAN code, it is often possible to write a different code which will also work. So if your FORTRAN code does not agree with the one given here, you are not necessarily wrong.

All solutions given here can be transferred to either punched cards or marked cards; that is, we have not used card columns beyond column forty.

CHAPTER I:

1. KILLR, K123, MAD: wrong starting letter;
   123GO, ****: not valid identifiers at all;
   DASTARD,PROFESSOR,LOTTERY; LOTTRY, BEWARE,
   WHATONEARTH: more than five characters;
   BE WARY: interior blank (acceptable to some compilers);
   G123, DOG: correct.

2. 35,000 35,000. 35,000.0: comma not permitted;
   35 -35: decimal point missing

3. a) Comment card;
   b) Columns 1 to 5, inclusive;
   c) Distinguishes between initial card of a statement (blank or zero) and continuation cards (characters other than blank or zero);
   d) Column seven;
   e) 26 alphabetic, 10 numeric characters;
   f) L, T, P, Z are alphabetic; 2, 9, 7 are numeric; (, =, *, +, - are special; %, ? are unacceptable.

4. 7860,12, 0.0000786012, -7860.12, -0.0000786012, 0.786012, -0.786012, 0.0786012, -7.86012, -0.0786012, 0.999999, 999999.

5. a) To translate from FORTRAN to machine language;
   b) Translation is done during the "compile phase", the translated program is obeyed during the "execution phase";
   c) Easier language, same on all machines.

6. Listing of input cards produced during the compile phase; WRITE statements lead to output during execution phase.

7. a) 3A illegal, % illegal; b) 3A, 5B illegal;
   c) Legal, sum of A3+B5; d) decimal points missing: 3., 5.
   e) Legal, sum of three times A and five times B;
   f) Illegal, "*" followed by "=";
   g) Legal, negative of 2.15 times A; same for (h)
   i) Illegal, decimal point missing from 3.0;
   j) Legal, A divided by three;
   k) See (i); l) Legal, three divided by A.

8.
```
    1 FORMAT (7E16.6)
      A=5.1
      B=12.3
      C=7.8
      VOLUM=A*B*C
      WRITE (3,1) A,B,C,VOLUM
      STOP
      END
```

9.
```
    1 FORMAT (7E16.6)
      A=5.1
      B=12.3
      C=7.8
      VOLUM=A*B*C
      AREA=2.0*(A*B+B*C+C*A)
      WRITE (3,1) A,B,C,VOLUM,AREA
      STOP
      END
```

10.
```
    1 FORMAT (7E16.6)
      WAGE=2.25
      HOURS=35.0
      GROSS=WAGE*HOURS
      TXDDC=GROSS/5.0
      SUDDC=GROSS/10.0
      PAY=GROSS-TXDDC-SUDDC
      WRITE (3,1) WAGE,HOURS,GROSS,TXDDC
     1 ,SUDDC,PAY
      STOP
      END
```

11. Same as 10 but the last four statements are replaced by:
```
      SVDDC=0.04*GROSS
      PAY=GROSS-TXDDC-SUDDC-SVDDC
      WRITE (3,1) WAGE,HOURS,GROSS,TXDDC
     1 ,SUDDC,SVDDC,PAY
      STOP
      END
```

12. Same as 11 but the fourth statement is replaced by:
```
      OVTIM=6.0
      GROSS=WAGE*HOURS+1.5*WAGE*OVTIM
```

CHAPTER II:

1. The "=" sign is a replacement symbol, not an equality. The quantity to the right of the "=" sign is evaluated, and stored into the memory location named to the left of the "=" sign.

2. a) Unknown b) 14.4
   c) 4.8 d) 0.0

3. a) C=A+B b) A=A+B
   c) C=(A-1.5)*B d) A=(B-1.5)*A
   e) C=A*B-1.5 f) C=A**2+B**3
   g) C=A**2*B**3 h) C=(A+B)*(A-B)

4. a) 40.0 b) 84.0
   c) 40.0 d) 9.0
   e) 9.0 f) 9.0
   g) 44.0 h) 108.0
   i) 1296.0 j) 1.33333...
   k) 16.0

5. a, d, g, h, i, k: mixed expressions
   b, c: expression appears on left side of equal sign;
   e, f, o, p: asterisk for multiplication missing;
   h, j: unmatched parentheses;
   m: adjacent operators ("*" and "-"),
   q: more than five characters in identifier name;
   l, n: grammatically correct.

6. a) S=9.801*T**2/2.0
   b) V=4.0*3.14159*R**3/3.0
   c) E=AI*R
   d) H=V**2/(2.0*9.801)
   e) V=(2.0*9.801*H)**0.5
   f) A=3.14159*R**2
   g) R=(A/3.14159)**0.5

CHAPTER III:

1. (b) is the only valid one

2. First statement valid, but silly (same as GO TO 68);
   second and third statements can not be reached in execution (no statement numbers, after transfer);
   third statement contains a mixed expression: Y-6;
   fourth statement: invalid statement number in GO TO ...
   fifth statement: never-ending loop;
   sixth statement: "STOP" is not a statement number.

3. a)
```
      U=C
      IF(F*(C-D)) 1200,1300,1300
1200  U=D
1300  ... (whatever comes next)
```
   b)
```
      TEST=D
      IF((B-2.5)*(3.5-B)) 1200,1200,1300
1200  TEST=C
1300  ... (whatever comes next)
```

4. a) Because of round-off errors, it is unlikely that B will ever equal 13.5 in the machine, exactly. Hence this is probably a never-ending loop.
   b) The statement B=A**2 can never be reached in execution.
   c) Infinite loop at statement 2300.

5.
```
a)    IF(A-2.0*B) 3000,3000,2000
b)    IF(A-2.0*B) 3000,2000,2000
c)    IF(A-2.0*B) 2000,3000,2000
d)    IF(A-2.0*B) 2000,3000,3000
e)    IF(A-2.0*B) 2000,2000,3000
f)    IF((A-2.0*B)*(3.0*B-A)) 3000,3000
     1 , 2000
g)    IF((A-2.0*B)*(3.0*B-A)) 3000,2000
     1 , 2000
h)    IF(X-A) 3000,3000,1100
1100  IF(X-B) 3000,3000,1200
1200  IF(X-C) 3000,3000,2000
i)    IF(X-A) 1100,1100,2000
1100  IF(X-B) 1200,1200,2000
1200  IF(X-C) 3000,3000,2000
j)    IF((X-1.6)*(2.4-X))  3000,3000,1100
1100  IF((Y-3.7)*(4.9-Y))  2000,3000,3000
k)    IF((X-1.6)*(6.9-X))  1100,3000,3000
1100  IF((Y-1.6)*(6.9-Y))  2000,3000,3000
l)    IF((X-A)**2+(Y-B)**2) 3000,
     1 2000,3000
```

6. a) Absolute value of A less than 3.1;
   b) Absolute value of A less than or equal to 3.1;
   c) Absolute value of A equal to 3.1;
   d) Absolute value of A greater than 3.1;
   e) Same as (a);
   f) A lies between 1.0 and 3.7.

CHAPTER IV:

1. Data within a FORTRAN program are handled by the FORTRAN compiler, in the compile phase, and appear permanently in the translated program. If no data are read in at execution time, then the program gives exactly the same output whenever it is run, and can therefore be run only once, usefully. Data read in at execution time are read under control of the (translated) READ statements in the object program; different data lead to different final results, with the same object program.

2. Data cards are not "understood" by the FORTRAN compiler. The data cards belong after the control card which signals the end of the compile phase.

3. Different installations use different numbers for these unit numbers. A program using NREAD and NPRNT needs to be modified at only one point: where NREAD and NPRNT are set. This point must come before the values of NREAD and NPRNT are used in the program.

4. Seven data numbers, four on first card (in fields 1, 2, 3, and 4, respectively) go into locations ALL, OF, US, and ARE, respectively; three data numbers on the second card (in fields 1, 2, and 3, respectively) go into locations VERY, CUTE, and GIRLS, respectively.

5. a) Closing bracket missing after "2"; LIFE and IS have wrong (integer) type; ENOUGH has more than five characters.
   b) FORMAT is not a statement number; LIFE and IS have wrong type.
   c) Attempt to read from the line printer.
   d) Wrong FORMAT statement number for printing a standard line of numbers; this may work in some cases, but will produce numbers in a different form from the E-form.
   e) Parentheses missing around "NPRNT,1".
   f) Parentheses missing around "NREAD,2".
   g) The second comma should be a right parenthesis.
   h) Delete the second comma.
   i) Delete the final comma.

6. a) Replace "25H" by "26H".
   b) Supply final right parenthesis.
   c) Delete the comma after "FORMAT".
   d) Replace "27H" by "26H", or else move the right parenthesis over towards the right.
   e) Every FORMAT statement must have a statement no.!

7. In the program on page 38, replace the two statements after statement 2000 by the new sequence:
```
      IF(WAGE-2.30) 2100,2100,2200
2100  TXDDC=0.12*GROSS
      GO TO 2500
2200  IF(WAGE-3.45) 2300,2300,2400
2300  TXDDC=0.15*GROSS
      GO TO 2500
2400  TXDDC=0.20*GROSS
2500  PAY=GROSS-TXDDC
```

8. In the program on page 31 replace the first four statements by:
```
      1 FORMAT (7E16.6)
      2 FORMAT (4F10.0)
        NREAD=1
        NPRNT=3
C       MODIFY THESE IF REQUIRED
 1000   READ (NREAD,2) A
        IF(A) 3000,3000,1100
 1100   TOLER=0.000001
        P=1.0
```
   also, the last three statements become:
```
 2500   WRITE (NPRNT,1) A,Q
        GO TO 1000
 3000   STOP
        END
```

9. In the modification given under (8), replace statement 1000 and the IF statement after it by the sequence:

```
      WRITE (NPRNT,900)
  900 FORMAT (9X,5HVALUE,7X,
     1 11HSQUARE ROOT/)
 1000 READ (NREAD,2) A
      IF(A) 1050,3000,1100
 1050 WRITE (NPRNT,1060)
 1060 FORMAT (18H NEGATIVE ARGUMENT)
      Q=0.0
      GO TO 2500
```

10. Replace the first four statements on page 25 by:

```
    1 FORMAT (7E16.6)
    2 FORMAT (4F10.0)
      NREAD=1
      NPRNT=3
C     MODIFY THESE IF REQUIRED
 1000 READ (NREAD,2) DEBT,PAYMT
      IF(DEBT**2+PAYMT**2)1200,1200,1500
 1200 STOP
 1500 CHARG=0.008*DEBT
```

Also, replace the statement STOP after 2500 by:

```
      GO TO 1000
```

11. In the modification above, replace statements 1000 to 1500 by:

```
 1000 READ(NREAD,2) DEBT,PAYMT
      IF(DEBT) 1050,1100,1300
 1050 WRITE (NPRNT,1060)
 1060 FORMAT (18H UNREASONABLE DATA)
      WRITE (NPRNT,1) DEBT,PAYMT
      GO TO 1000
 1100 IF(PAYMT) 1050,1200,1050
 1200 STOP
 1300 IF(PAYMT-0.008*DEBT)1050,1050,1400
 1400 WRITE (NPRNT,1450)
 1450 FORMAT (10X,4HDEBT,10X,6HCHARGE,9X
     1 ,7HPAYMENT,7X,9HREDUCTION,7X,
     2 9HREMAINDER/)
 1500 CHARG=0.008*DEBT
```

Without the test of statement 1300, we could get into a situation where the payment is too small, so that the debt increases (rather than decreases) each month, and is never paid off. Result: never-ending loop!

CHAPTER VI:

1. "Mixed Mode" means an expression requiring addition, subtraction, multiplication, or division of two numbers of different types (integer and real). There are no machine commands for this, and USA Standard FORTRAN forbids mixed mode expressions.

2. A FORTRAN constant stays the same throughout program execution. A FORTRAN variable is the name of a memory location in the machine, which contains a number which may alter during program execution.

3. c, f: decimal point is incorrect; use e instead of f;
d: omit interior comma;
a, b, e: correct.

4. a, g: missing decimal point;
e: superfluous comma, missing decimal point;
f, l: superfluous comma;
h: an exponent must follow the "E"; use 3.E0;
j: decimal point missing after 3;
k: see (h) and (j); correct form is 3.E0;
all the others are correct.

5. 5.27 5270. 0.00527 0.00527 5270. 0.99999

6. a) Missing decimal point: 578.E-5
b) Wrong exponent: .578E+2
c) Missing sign: -.243E+3
d) Missing decimal point: 243.
e) Missing number of shifts: 75.9E0, or just 75.9
f) Correct

7. NUMBER, LOTTERY, MYGIRL: too many characters
BOY, PAPA: wrong starting letter
MY MAN: interior blank not permitted with some FORTRAN compilers, O.K. with others
MAN: special character "" not permitted
JACK, MAMA, MYNAH: correct

8. b, c, and e involve implied conversions; c gives 3, not 4, for the value of KLOT.

9. a) -2 b) -5 c) -2 d) -5

10. a) mixed mode b) 3
c) mixed code d) 2191.0
e) 17.0 f) 132
g) undefined h) 18.0
i) mixed mode j) undefined, needs parentheses
k) 17.0 l) mixed mode

11. Statement 1100: (integer)**(real) forbidden
Statement 1300: (zero)**(negative) forbidden
Statement 1500: (negative)**(real) forbidden
Statement 1600: (zero)**(negative) forbidden

CHAPTER VII:

1. a) The output list in WRITE (*n,m*) *List*
b) *n* is the unit number of the printer
c) *m* is the statement number of the FORMAT statement

2. a) Omit the comma after 1001
b) Omit the comma before QUO
c) Omit the comma after WILLY
d) The output list must consist of memory location names, not of actual numbers (constants)
e) B**2 is not a valid list element.

3. (We omit the statement number and the word "FORMAT")
a) (10I12) b) (I10,5E22.6)
c) (2I5,3F10.0) d) (I5,F15.0,I5,E15.0,4I5,F10.0)

4. a) Excessive total record length (432 characters)
b) Omit the comma after the word "FORMAT"
c) Replace comma by left parenthesis, supply a right par.
d) Replace comma by left parenthesis.
e) The field descriptor F10.9 is invalid for output, since it leaves no room for a digit before the decimal point. The statement may work for input.
f) The field descriptor E12.6 is invalid for output, since it specifies insufficient field width; may work for input.

5. a)

```
      FORMAT (I15,3E25.6/15X,4F25.3)
```

b)

```
      FORMAT (14H NEW ITERATION //3H K=,
     1I15//3H A=,E25.6,8X,2HB=,E25.6,8X,
     22HC=,E25.6//3H D=,F25.3,8X,2HE=,
     3 F25.3,8X,2HF=,F25.3/3H G=,F25.3/)
```

6. a) Integer in field of width seven (first character of the I8 field is not printed), then five real numbers, each in a field of width twelve, each with two digits after the decimal point, right-adjusted in each field.

b)

```
      WRITE (NPRNT,1450)
1450 FORMAT (3X,5HMONTH,8X,4HDEBT,6X,
     1 6HCHARGE,5X,7HPAYMENT,3X,
     2 9HREDUCTION,3X,9HREMAINDER/)
```

7. In the solution to problem 11 of Chapter IV (see page 155) make the following additional changes:
   i) Replace statement 1450 by statement 1450 at the bottom right of page 155 (solution to 6b);
   ii) Replace statement 1500 by the three statements:
```
        MONTH=0
1500    MONTH=MONTH+1
        CHARG=0.008*DEBT
```
   iii) In the original program on page 25, replace statement number 2000 by:
```
2000  WRITE (NPRNT,2001) MONTH,DEBT,
     1 CHARG,PAYMT,REDUC,RMAIN
2001  FORMAT (I8,5F12.2)
```
   iv) Replace statement 2500 by:
```
2500  WRITE (NPRNT,2001) MONTH,DEBT,
     1 CHARG,PAYMT
```

8.
```
      NPRNT=3
      WRITE (NPRNT,1000)
1000  FORMAT (4X,6HNUMBER,4X,6HSQUARE,6X
     1 ,4HCUBE,4X,6HFOURTH/)
      NUMB=1
C     LOOP STARTS HERE
1500  KWAR=NUMB**2
      KUBE=KWAR*NUMB
      KOUR=KUBE*NUMB
      WRITE (NPRNT,1600) NUMB,KWAR,KUBE,
     1 KOUR
1600  FORMAT (4I10)
C     NOW ADVANCE NUMB AND TEST FOR END
1800  NUMB=NUMB+1
2000  IF(NUMB-10) 1500,1500,2500
C     ALL DONE
2500  STOP
      END
```

CHAPTER VIII:

1.
```
      NREAD=1
1100  READ (NREAD,1200) X,INDEX
1200  FORMAT (E10.0,I5)
      IF(INDEX*(6-INDEX) 1400,1400,1300
1300  GO TO (1301,1302,1303,1304,1305),
     1 INDEX
1301  A1=X
      GO TO 1100
1302  A2=X
      GO TO 1100
1303  A3=X
      GO TO 1100
1304  A4=X
      GO TO 1100
1305  A5=X
      GO TO 1100
C     WE GET HERE AFTER INDEX IS NOT ONE
C          OF 1, 2, 3, 4, OR 5
1400  ... (whatever comes next)
```

2. a) Omit the comma before I
   b) K+5 not permitted; use:
```
      KAND5=K+5
      DO 3500 J=1,KAND5
```
   c) B is not of type integer
   d) Omit the TILL
   e) Omit the FOR
   f) Omit the comma at the very end
   g) R is not of type integer
   h) The name J is used both for the loop variable and for the termination test. This is inconsistent, since updating the loop variable J would also update (alter) the termination test during operation of the loop.

3. a) -1 in first line is illegal
      mixed mode expression in second line
      attempt to alter loop variable J in third line
      attempt to jump into the range of the DO loop in line 5
   b) Last statement of first DO loop (statement 1000) must not itself be a DO statement.
   c) Same name "I" used for loop variable and for increment
      Last statement of a DO loop (statement 1000) must not be an IF statement.
      Line 5: Attempt to use value of I after normal completion of a DO loop on I; I is undefined at this point.
      Line 7: Attempt to jump into the range of a DO loop, from outside that loop.

4. a, d are valid, but none of the others.

CHAPTER IX:

1. a) A*SIN(U)-B
   b) A*SIN(U-B)
   c) SIN(SQRT(ALOG(A**2-B**3)))
   d) COS(EXP(U)+SQRT(W))
   e) TANH(ATAN(U)+FLOAT(K))
   f) SIN((X-FLOAT(M))/(X-FLOAT(2*M)))

2. a) COTAN(A)=COS(A)/SIN(A)
   b) SINH(A)=0.5*(EXP(A)-1.0/EXP(A))
      Note: This is likely to give very poor accuracy if A is much smaller than one, in absolute value. For this reason, a FUNCTION subprogram would be much better.
      Second Note: The use of 1.0/EXP(A) is better than the alternative EXP(-A), since most FORTRAN compilers will recognize that EXP(A) occurs twice, and will enter the exponential function routine only once, thereby saving machine time.
   c) COSH(A)=0.5*(EXP(A)+1.0/EXP(A))

3. a) Statement function definition must not come later than an executable statement (I=1).
   b) Line 3: Integer argument for SIN is illegal.
      Line 4: EXPSQ used without any argument.
   c) (Bet you didn't find it!) Unmatched parentheses in line 1 (final right parenthesis is missing).

CHAPTER X:

1. Array A contains the numbers: -2.0, -2.0, 0.0, 4.0, 10.0
   Array LOON contains the numbers: 2, 1, 2, 3, 4
   Array B contains the numbers: 3.0, 5.0, 7.0, 9.0, unknown.
   The array element B(2) is set twice during the loop (for I=1 and for I=3), the second time giving the value quoted here. The array element B(5) is not set at all, since J is never equal to 5.

2. a)
```
      TEMP=G(15)
      DO 1200 I=1,14
      J=15-I
      G(J+1)=G(J)
1200  CONTINUE
      G(1)=TEMP
```
   Note: You should make sure that you understand why we could not have written simply G(I+1)=G(I) within the DO-loop.
   b)
```
      TEMP=G(N)
      K=(N+1)/2
      G(N)=G(K)
      G(K)=TEMP
```
   c)
```
      DO 1200 I=1,19,2
      TEMP=C(I)
```

(over)

```
(cont.)   C(I)=C(I+1)
          C(I+1)=TEMP
   1200   CONTINUE
```

3.
```
          DIMENSION KOOKY(8,8)
          DO 1500 I=1,8
          DO 1400 J=1,8
          KOOKY(I,J)=0
          IF(I-2*(I/2)) 1400,1100,1200
C         I IS EVEN
   1100   IF(J-2*(J/2)) 1400,1400,1150
   1150   KOOKY(I,J)=I-J
          GO TO 1400
C         I IS ODD
   1200   IF(J-2*(J/2)) 1400,1300,1400
   1300   KOOKY(I,J)=I+J
   1400   CONTINUE
   1500   CONTINUE
```

4. Preface all programs with: DIMENSION A(13),B(13)

a)
```
          DO 1100 I=1,13
          B(I)=Q*A(I)
   1100   CONTINUE
```

b)
```
          DO 1100 I=1,13
          A(I)=A(I)+B(I)
   1100   CONTINUE
```

c)
```
          DO 1200 I=1,13
          B(I)=A(I)-B(I)
   1200   CONTINUE
```

d)
```
          Q=0.0
          DO 1300 I=1,13
          Q=Q+A(I)*B(I)
   1300   CONTINUE
```

e)
```
          Q=0.0
          DO 1400 I=1,13
          Q=Q+A(I)**2
   1400   CONTINUE
          Q=SQRT(Q)
```

5.
```
          DIMENSION A(100)
          WHY=1.0
          DO 1100 I=1,N
          WHY=WHY *A(I)
   1100   CONTINUE
```

6.
```
          DIMENSION B(100)
          JOKE=M/2
          DO 1100 I=1,JOKE
          J=M+1-I
          TEMP=B(I)
          B(I)=B(J)
          B(J)=TEMP
   1100   CONTINUE
```

7. Preface all programs with: DIMENSION MORDE(8,8)

a)
```
          DIMENSION MERDE(64)
          EQUIVALENCE (MORDE(1,1),MERDE(1))
          KOUNT=0
          DO 1100 I=1,64
          IF(MERDE(I)-2) 1100,1050,1100
   1050   KOUNT=KOUNT+1
   1100   CONTINUE
```

b) Same as (a), but replace the IF statement by:
```
          IF(MERDE(I)+1) 1100,1050,1100
```

c)
```
          KOUNT=0
          DO 1200 I=1,8
          IF(MORDE(I,I)-2) 1200,1100,1200
   1100   KOUNT=KOUNT+1
   1200   CONTINUE
```

d)
```
          KOUNT=0
          DO 1300 I=2,8
(cont.)   IMIN1=I-1
          DO 1200 J=1,IMIN1
          IF(MORDE(I,J)+2) 1200,1100,1200
   1100   KOUNT=KOUNT+1
   1200   CONTINUE
   1300   CONTINUE
```

e)
```
          KOUNT=0
          DO 1300 I=1,8
          DO 1200 J=1,8
          IF(MORDE(I,J)-1) 1200,1100,1200
   1100   IF(I-J) 1150,1200,1150
   1150   KOUNT=KOUNT+1
   1200   CONTINUE
   1300   CONTINUE
```

f) We assume that black pieces move towards smaller I.
```
          KOUNT=0
          DO 3600 I=3,8
          IP=I-1
          IPP=I-2
          DO 3500 J=1,8
          IF(MORDE(I,J)+1) 3500,3100,3500
C         THERE IS A BLACK PIECE AT (I,J)
   3100   DO 3400 K=1,3,2
          L=K-2
C         THIS L=-1 FOR JUMP TO LEFT
C               =+1 FOR JUMP TO RIGHT
          JP=J+L
          JPP=J+2*L
          IF(JPP*(9-JPP)) 3400,3400,3200
C         (IPP,JPP) IS ON THE CHECKER BOARD
C         TEST WHETHER IT IS EMPTY, AND
C         WHETHER (IP,JP) CONTAINS EITHER A
C         WHITE PIECE OR A WHITE KING
   3200   M=MORDE(IP,JP)
          N=MORDE(IPP,JPP)
          NN=IABS(N)
          MM=IABS((M-1)*(M-2))
          IF(NN+MM) 3400,3300,3400
C         ZERO IF AND ONLY IF (I,J) CAN JUMP
   3300   KOUNT=KOUNT+1
C         NOW GO OUT LIKE A FLASH
C             WE ARE TO COUNT PIECES THAT CAN
C             JUMP, NOT NO. OF POSSIBLE JUMPS.
          GO TO 3500
   3400   CONTINUE
   3500   CONTINUE
   3600   CONTINUE
```

8. There are eight data cards, each containing eight numbers in fields of width three, each. Each card gives a column of the checker board, reading from the top down.

9. The printed page would give eight lines, the first line containing information about the first column (not the first row!) of the checker board, the second line containing information about the second column (!) of the checker board, and so on. This amounts to a scrambled version of what the checker board really looks like.

10.
```
          DIMENSION KROW(8),MORDE(8,8)
          NREAD=1
          READ (NREAD,1100) I,KROW
   1100   FORMAT (9I3)
          DO 1200 J=1,8
          MORDE(I,J)=KROW(J)
   1200   CONTINUE
```

11.
```
          DIMENSION KROW(8),MORDE(8,8)
          NREAD=1
   1000   READ (NREAD,1100) I,KROW
   1100   FORMAT (9I3)
```

(over)

```
(cont.)  IF(I) 1300,1500,1150
  1150   DO 1200 J=1,8
         MORDE(I,J)=KROW(J)
  1200   CONTINUE
         GO TO 1000
  1300   L=-I
         DO 1400 K=1,8
         MORDE(K,L)=KROW(K)
  1400   CONTINUE
         GO TO 1000
  1500   ... (whatever comes next)
```

```
12.      DIMENSION JOKER(8),MORDE(8,8)
         NPRNT=3
         DO 1300 I=1,8
         DO 1100 J=1,8
         JOKER(J)=MORDE(I,J)
  1100   CONTINUE
         WRITE (NPRNT,1200) JOKER
  1200   FORMAT (8I8///)
  1300   CONTINUE
```

```
13.a)    DIMENSION B(11,11),VEC(11)
         EQUIVALENCE (B(1,1),VEC(1))
```

```
   b)    DIMENSION B(11,11),VEC(11)
         EQUIVALENCE (B(1,11),VEC(1))
```

c) The elements of the second row of B are not next to each other in machine storage, whereas the elements of VEC are adjacent in machine storage.

```
   d)    DIMENSION CARDS(4,13),QUEEN(4),
        1 QUING(4),ACES(4)
         EQUIVALENCE (CARDS(1,11),QUEEN(1))
         EQUIVALENCE (CARDS(1,12),QUING(1))
         EQUIVALENCE (CARDS(1,13),ACES(1))
```

14. We preface each list of errors with the statement number
1000: Second dimension of NOHOW must not be 0.
1100: Negative dimension for VIC;
WAG is much too big to fit into memory;
WOG has four subscripts (more than two);
1200: Adjustable dimension K not permitted;
DIMENSION statement must not come after some executable statements;
1500: I-field descriptor for reading real numbers is wrong;
1600: Illegal subscript expression (3*I+J);
1700: YAHOO is a one-subscript (vector) array;
1800: Subscript I-2 has forbidden value zero;
1900: Illegal subscript expression (35-I);
2100: Subscript overflow since K=8 exceeds the declared dimensions of YA;
2200: Also subscript overflow; note that both statements 2100 and 2200 may be obeyed in execution, thereby overwriting two perfectly innocent storage locations.
2300: Control transfer to non-executable statement;
2400: Can never be reached in program execution.

CHAPTER XI:

```
1.       WRITE (NPRNT,1100) (J,J=1,6),
        1 (I,(B(I,J),J=1,6),I=1,5)
  1100   FORMAT (20X,14HMATRIX ARRAY B//
         17H ROW NO,6(7X,6HCOLUMN,I2)//(I7,
        2 6E15.5))
```

```
2.       WRITE (NPRNT,1200)(I,F(I,I),I=1,50)
  1200   FORMAT (23H DIAGONAL VALUES F(I,I)
         1//(I10,E15.5,I10,E15.5,I10,E15.5,
        2 I10,E15.5))
```

3. a) Ten lines are printed. Line number I contains the value of I (field I4), the value of MACK(I) in a field of width ten, then five blank spaces, then the I'th row of the matrix JACK, which means three integers, each in a field of width eight. Only the first half of MACK and of JACK are printed out.

b) We abbreviate MUTT to M, JEFF to J, and print the subscripts as real "subscripts", rather than within parentheses. There are six lines altogether, containing the table layout:

3 $M_{3,4}$ $J_{4,3}$ $M_{3,6}$ $J_{6,3}$ $M_{3,8}$ $J_{8,3}$ $M_{3,10}$ $J_{10,3}$
$M_{3,12}$ $J_{12,3}$ $M_{3,14}$ $J_{14,3}$ $M_{3,16}$ $J_{16,3}$ $M_{3,18}$ $J_{18,3}$
blank line

8 $M_{8,4}$ $J_{4,8}$ $M_{8,6}$ $J_{6,8}$ $M_{8,8}$ $J_{8,8}$ $M_{8,10}$ $J_{10,8}$
$M_{8,12}$ $J_{12,8}$ $M_{8,14}$ $J_{14,8}$ $M_{8,16}$ $J_{16,8}$ $M_{8,18}$ $J_{18,8}$
blank line

```
4.       WRITE (NPRNT,1300)
        1 (I,(C(I,J),J=1,9),I=1,9,2)
  1300   FORMAT (23H ODD-NUMBERED ROWS OF C
         1//(I10,5E16.6/10X,4E16.6/))
```

```
5.       WRITE (NPRNT,1400)
        1 (I,(G(I,J),J=1,24),I=N,M,2)
  1400   FORMAT (15H SOME ROWS OF G//(I10,
        1 6E16.6/10X,6E16.6/10X,6E16.6/10X
        2 ,6E16.6/))
```

```
6.       WRITE (NPRNT,1500)
  1500   FORMAT (14H ELEMENTS OF H/
         127H ON AND BELOW MAIN DIAGONAL/)
         DO 1700 I=1,25
         WRITE (NPRNT,1600) I,(H(I,J),J=1,I)
  1600   FORMAT (I10,5E16.6/(10X,5E16.6))
  1700   CONTINUE
```

```
7. a)    DIMENSION MORDE(8,8)
         READ (NREAD,1700)
        1 ((MORDE(I,J),J=1,8),I=1,8)
  1700   FORMAT (64I1)
         DO 1900 I=1,8
         DO 1800 J=1,8
         MORDE(I,J)=MORDE(I,J)-2
  1800   CONTINUE
  1900   CONTINUE
```

b) We assume that the array MORDE must not be destroyed during the punching of the card. Hence we introduce another array MERDE as a buffer.

```
         DIMENSION MORDE(8,8),MERDE(64)
         NPNCH=2
C        THIS IS THE UNIT NUMBER OF THE CARD
C        PUNCH ATTACHED TO THE COMPUTER
         DO 2100 I=1,8
         DO 2000 J=1,8
         K=J+8*(I-1)
C        THIS IS ROW SEQUENCE NO. OF (I,J)
         MERDE(K)=MORDE(I,J)+2
  2000   CONTINUE
  2100   CONTINUE
         WRITE (NPNCH,2200) MERDE
  2200   FORMAT (64I1)
```

```
8.       DIMENSION D(15,15),E(225)
         EQUIVALENCE (D(1,1),E(1))
         NPNCH=2
C        THIS IS THE UNIT NUMBER OF THE CARD
C        PUNCH ATTACHED TO THE COMPUTER.
C        ALTER IF NEEDED FOR YOUR COMPUTER.
         IT=I/10
         IF(I-10*IT)  2800,2300,2800
C        I IS DIVISIBLE BY 10
```

(over)

```
      2300 DO 2600 N=1,45
      C    N = NO. OF CARD BEING PUNCHED
           L=5*(N-1)+1
           M=5*N
      C    THESE ARE ORDER NOS. WITHIN E
           WRITE (NPNCH,2500) (E(K),K=L,M)
          1 ,IT,N
      2500 FORMAT (5E14.7,2X,3HSCR,I3,I2)
      2600 CONTINUE
      2800 ... (whatever comes next)
```

9.
```
           READ (NREAD,2900)
          1 (V(I,5),V(I,6),I=1,5)
      2900 FORMAT (2E10.0)
```

10.
```
           READ (NREAD,3000)
          1 ((YAHOO(I,J),J=1,19),I=2,14,2)
      3000 FORMAT (7E10.0)
```

11. When reading restart information from data cards (rather than magnetic tape), it is good practice to read initially into temporary locations. Then we carry out tests to ensure that we have been reading the right card. Only after these tests have succeeded, do we transfer from temporary to permanent locations.

```
           DIMENSION D(15,15),E(225),VEC(5)
           EQUIVALENCE (D(1,1),E(1))
           DO 4000 KARD=1,45
           READ (NREAD,3200) VEC,JT,N
      3200 FORMAT (5E14.7,5X,I3,I2)
           IF(KARD-1) 3300,3300,3400
      3300 IT=JT
      3400 LADY=IABS(IT-JT)
           LOUT=IABS(N-KARD)
           IF(LADY+LOUT) 3700,3900,3700
      C    A BAD CARD HAS BEEN READ
      3700 WRITE (NPRNT,3800) VEC,JT,N
      3800 FORMAT (4H BAD,5E14.7,5X,I3,I2)
           STOP
      C    THE CARD HAS PASSED THE TESTS
      3900 DO 3950 KOD=1,5
           KID=KOD+5*(KARD-1)
           E(KID)=VEC(KOD)
      3950 CONTINUE
      4000 CONTINUE
```

12.
```
           DIMENSION D(15,15)
           NTAPE=8
           IOTEN=1/10
      C    DOES 10 DIVIDE INTO I ?
           IF(I-10*IOTEN) 4400,4100,4400
      C    YES, IT DOES. START DUMP
      4100 WRITE (NPRNT,4200) I,NTAPE
      4200 FORMAT (23H STARTING DUMP, I,TAPE=
          1 ,2I10)
           WRITE (NTAPE) IOTEN,D
      C    NOTE THAT IOTEN IS THE RECORD NO.
           WRITE (NPRNT,4300)
      4300 FORMAT (14H DUMP COMPLETE )
      4400 ... (whatever comes next)
```

13.
```
           DIMENSION D(15,15),E(225),DD(225)
           EQUIVALENCE (D(1,1),E(1))
           NTAPE=8
           REWIND NTAPE
      4450 READ (NREAD,4500) I
      4500 FORMAT (I5)
           IOTEN=I/10
           IF(I-10*IOTEN) 4600,4700,4600
      4600 WRITE (NPRNT,4650) I
      4650 FORMAT (3H I=,I10,
          1 20H NOT DIVISIBLE BY 10 )
           STOP
      C    I IS O.K., IOTEN = REC.NO.
      4700 NSKIP=IOTEN-1
      C    COUNT OF RECORDS TO BE SKIPPED
           IF(NSKIP) 5000,5000,4800
      4800 DO 4900 NREC=1,NSKIP
           READ (NTAPE) KREC
           IF(NREC-KREC) 4850,4900,4850
      4850 WRITE (NPRNT,4860) NREC,KREC
      4860 FORMAT (8H REC.NO.,I10,
          1 11H INSTEAD OF,I10)
           STOP
      4900 CONTINUE
      C    TAPE IS PROPERLY POSITIONED
      5000 READ (NTAPE) IOTON,DD
           IF(IOTON-IOTEN) 5100,5300,5100
      5100 WRITE (NPRNT,4860) IOTON,IOTEN
           STOP
      C    ALL CLEAR, DISTRIBUTE THE LOOT
      5300 DO 5500 I=1,225
           E(I)=DD(I)
      5500 CONTINUE
```

14.
```
           DIMENSION X(3),XP(3)
           NTAPE=7
           KREC=0
           DIST=0.0
           REWIND NTAPE
      1000 KREC=KREC+1
           READ (NTAPE) NREC,X
           IF(NREC-KREC) 1100,1200,1100
      1100 WRITE (NPRNT,1150) NREC,KREC
      1150 FORMAT (13H REC.NO.WRONG,2I10)
           STOP
      1200 T=0.0
           DO 1400 I=1,3
           IF(NREC-1) 1300,1300,1250
      1250 T=T+(X(I)-XP(I))**2
      1300 XP(I)=X(I)
      1400 CONTINUE
           DIST=DIST+SQRT(T)
      1500 IF(X(3)) 1700,1000,1000
      1700 WRITE (NPRNT,1750) DIST
      1750 FORMAT (13H PATH LENGTH=,F20.6)
           REWIND NTAPE
           STOP
           END
```

15. Make the following changes in the preceding code:

```
           DIMENSION X(3,20),XP(3,20)
```

Replace statements 1200 to 1500 by:

```
      1200 DO 1450 J=1,20
           T=0.0
           DO 1400 I=1,3
           IF(NREC+J-2) 1300,1300,1250
      1250 T=T+(X(I,J)-XP(I,J))**2
      1300 XP(I,J)=X(I,J)
      1400 CONTINUE
           DIST=DIST+SQRT(T)
           IF(X(3,J)) 1700,1450,1450
      1450 CONTINUE
      1500 GO TO 1000
```

CHAPTER XII:

1. a) The FUNCTION statement;
   b) To return control to the calling program;
   c) The function name appears to the left of the equal sign in an assignment statement;
   d) The first letter of the function name.

2. Y= 22.50

3.
```
0.170000E 01  FITS IN POSITION    0
0.984000E 02  FITS IN POSITION    3
0.380000E 01  FITS IN POSITION    1
```

4. a)
```
      FUNCTION ZIGGY(X)
      Y=ABS(X)
      K=Y/2.0
      Z=Y-FLOAT(2*K)
      ZIGGY=1.0-ABS(Z-1.0)
      RETURN
      END
```

b)
```
      FUNCTION SQWVE(X)
      Y=ABS(X)
      K=Y
      IF(Y-FLOAT(K) 1100,1100,1200
1100  SQWVE=0.0
      GO TO 1600
1200  IF(K-2*(K/2)) 1300,1300,1400
1300  SQWVE=SIGN(1.0,X)
      GO TO 1600
1400  SQWVE=-SIGN(1.0,X)
1600  RETURN
      END
```

c)
```
      FUNCTION SMALL(ARRAY,N)
      DIMENSION ARRAY(100)
      T=ARRAY(1)
      IF(N-1) 1400,1400,1100
1100  DO 1300 K=2,N
      IF(ARRAY(K)-T) 1200,1300,1300
1200  T=ARRAY(K)
1300  CONTINUE
1400  SMALL=T
      RETURN
      END
```

d)
```
      FUNCTION TRACE(ATRIX)
      DIMENSION ATRIX(10,10)
      T=0.0
      DO 1100 I=1,10
      T=T+ATRIX(I,I)
1100  CONTINUE
      TRACE=T
      RETURN
      END
```

5. 6 2 4 16

6.
```
      SUBROUTINE TRNSP(ATRIX,N)
      DIMENSION ATRIX(10,10)
      IF(N-1) 1600,1600,1100
C     NOTE THIS TEST WAS NECESSARY.
C     N EXCEEDS 1.  GO TO IT.
1100  DO 1500 I=2,N
      IMIN1=I-1
      DO 1400 J=1,IMIN1
      T=ATRIX(I,J)
      ATRIX(I,J)=ATRIX(J,I)
      ATRIX(J,I)=T
1400  CONTINUE
1500  CONTINUE
1600  RETURN
      END
```

7.
```
      SUBROUTINE PUTIN(X,K,VEC,N)
      DIMENSION VEC(100)
      MOVE=N-K+1
C     THIS IS NO. OF ITEMS TO BE MOVED
      IF(MOVE) 1300,1300,1100
1100  DO 1200 I=1,MOVE
      MONK=N+1-I
C     EXPLAIN WHY WE MOVE IN THIS ORDER.
      VEC(MONK+1)=VEC(MONK)
1200  CONTINUE
1300  VEC(K)=X
      N=N+1
      RETURN
      END
```

8.
```
      SUBROUTINE REMOV(K,VEC,N)
      DIMENSION VEC(100)
      MOVE=N-K
      IF(MOVE) 1300,1300,1100
1100  DO 1200 I=1,MOVE
      MONK=K+I-1
      VEC(MONK)=VEC(MONK+1)
1200  CONTINUE
1300  VEC(N)=0.0
      N=N-1
      RETURN
      END
```

9. a) 2.0 3.0 unknown
   b) 15.0 3.0 5.0
   c) 30.0 3.0 5.0
   d) 3267.0

10. a) 5
    b) 169
    c) 120
    d) 179

PART C

In this, third, part of the book, we present material relevant to your particular installation, that is, the methods you must employ to prepare input cards for your jobs; the control cards required for a proper job on your machine and operating system; the diagnostic messages and execution time error messages put out by your FORTRAN compiler and operating system; and the special features of your version of the FORTRAN language.

This particular version of PART C deals with the FORTRAN compiler called MIDITRAN, which forms part of the Monash University Educational Computer System (ECS). This system was developed and is maintained by the Monash University Computer Centre at Monash University, Clayton, Victoria, Australia 3168. It is especially designed for student teaching, and contains other languages besides FORTRAN (for example, there is an ECS COBOL compiler as well). Most features of MIDITRAN are computer-independent; the few computer-dependent aspects are listed in Chapter XVII, section A. I am grateful to the staff of the Monash University Computer Centre for a great deal of help and cooperation in the preparation of this version of PART C.

TABLE OF CONTENTS FOR PART C

CHAPTER XIV

CARD PREPARATION OF MIDITRAN CARDS

Section A: Description of the Cards.

A typical MIDITRAN card is shown on page 10. There are two types: (1) optically marked cards, where marks are made with a soft pencil, and (2) "port-a-punch" cards, in which areas on the card corresponding to possible holes are loose enough so they can be pressed out with the sharp end of a bent paper clip (such card areas are called "chads").

Port-a-punch cards are the less desirable choice, but if you have to use them, observe the following precautions: (a) To press out a chad, lay the card flat on a table, put the end of the bent paper clip firmly on top of the desired position, then raise the card so that the chad is removed cleanly and completely; (b) *don't* use anything broader than a paper clip (e.g., a pencil), since broadened holes may be read incorrectly; (c) *don't* bend or fold the cards, for the same reason; (d) *don't* try to fix a card once an incorrect hole has been punched; attempts to stick a chad back into the hole are not only unreliable, but may lead to permanent damage to the card reader of the computer.

With optically marked cards (the preferred type) the *precautions* are: (a) Use a soft pencil; (b) Keep the marks straight and clean; it is not necessary, nor desirable, to widen the pencil mark so as to fill in the entire area between the two vertical lines – doing this can lead to smudging and resultant read errors; (c) It *is* desirable to make your pencil mark extend *vertically* at least as far as the vertical lines, preferably overshooting a bit both at the top and at the bottom, otherwise, the mark may be overlooked by the card reader; (d) Do *not* make pencil marks in odd positions on the card accidentally – such marks can result in misreading; an exception is the top edge of the card, which can and should be written on, see the example on page 10: the top edge is ignored by the card reader; (e) Erasing of marks, to correct a card, is possible, but requires considerable care; if done at all, it should be done with a plastic (vinyl) eraser, *not* with a rubber eraser; anything except an absolutely perfect erasure is likely to cause misreading.

Warning: Instructions (b) and (c) above are somewhat dependent on the particular card reader used at your installation. We recommend that you ask your computer centre whether these instructions apply as they stand, or need modification.

With either optically marked or port-a-punch cards, start by writing at the top of every column the character to be marked (or punched) into that column. There are three separate areas on the card shown on page 10:

1. The first three columns are reserved for a statement number (*up to three* digits only!) *or* a "comment" indicator (equivalent to a "C" in column 1 of a punched card), *or* a "cont" (continue) mark, equivalent to a non-blank character in column 6 of a punched card.

2. Area II of the card is for statement identification, and allows simple marking for FORTRAN keywords such as "GO TO"; for system control keywords of the ECS system, namely "*JOB,", "*DATA", and "*END"; and for the "card-piecing together" mark "EXTEND".

3. Area III of the card is the "main body"; it consists of sixteen sub-areas, labelled 1 to 16 along the bottom edge of the card. Each sub-area is used to represent exactly one character from the FORTRAN character set. (The blank character is indicated by a sub-area without pencil markings.) Some characters are represented by single marks; others, namely those for which the character symbol is printed halfway between two marking positions, are represented by pairs of marks, one just above and one just below the printed symbol.

Section B: FORTRAN program cards and ECS system control cards.

A typical FORTRAN program card is shown in Figure 3 on page 10. The statement number appears in area I, the keyword "FORMAT(" is obtained by a single mark in area II, and the remainder of the statement, namely "7E16.6)" is marked into area III.

The restriction to at most three digits in a statement number requires minor changes to programs shown in this book, to alter four-digit to three-digit statement numbers. Just omit the final zero of every statement number, systematically. For example, on page 22, statement number 1500 becomes statement number 150 with MIDITRAN cards; *and* "GO TO 1500" is changed to "GO TO 150".

The Basic FORTRAN IV keywords available in area II of the card are: "GO TO", "DO", "CALL", "STOP", "READ(", "WRITE(", "SUBROUTINE", "FUNCTION", "COMMON", "DIMENSION", "IF(", "CONTINUE", "RETURN", "PAUSE", "FORMAT(", and "END". Full FORTRAN keywords also available are: "INTEGER", "REAL", "EXTERNAL", and "DATA". Their use is explained in Chapter XVII, section D. Non-standard FORTRAN keywords are "READ" (*without* a left parenthesis following it) and "PRINT". These are used for free-format I/O (see Chapter XV, section B) and FORTRAN II type I/O (see Chapter XVII, section D).

Certain Basic FORTRAN keywords are missing, namely, REWIND, BACKSPACE, ENDFILE, and EQUIVALENCE. These FORTRAN features are not in MIDITRAN and can not be used. (See Chapter XVII, section B, for restrictions in MIDITRAN.)

FORTRAN keywords available in area II of the card should *always be used.* Do *not* write out "GO TO" in area III of the card, but rather use the "GO TO" mark in area II! Failure to do so may lead to diagnostic message 12 ("unidentifiable statement") from the compiler.

Do *not* repeat, in area III, characters already implied by a mark in area II. For example, on page 10, the left parenthesis "(" is already implied by the marked position "FORMAT(" in area II, and therefore must not appear again in area III. Repetition of the "(" in area III results in diagnostic error message 37 ("brackets do not match"), since the compiler would read such a card as

```
1 FORMAT((7E16.6)
```

with two left brackets but only one right bracket. The same caution applies to "READ(", "WRITE(", and "IF(".

Do *not* use the position "READ" in area II if you want a *standard* FORTRAN input command of the type discussed in Chapter IV; use the "READ(" position instead! "READ" without the left bracket is reserved for non-standard FORTRAN input commands.

Since area III can accommodate at most sixteen separate characters, a single FORTRAN line may fail to fit onto one MIDITRAN card. In that case, more than one MIDITRAN card may be used; the leading card contains the statement number (if any) in area I, the FORTRAN keyword (if any) in area II, and sixteen characters from the FORTRAN line in area III. The next MIDITRAN card is appended as follows: area I is blank, area II is marked in the "EXTEND" position, and the next sixteen characters from the FORTRAN line are placed into area III. Further "EXTEND" cards may follow, as many as are needed to go up to column 72 of an ordinary punched FORTRAN statement card. For instance, the FORTRAN statement:

```
170 IF((WAGE-1.50)*(4.25-WAGE)) 180,190,190
```

is spread over three MIDITRAN cards, a leading card and two EXTEND cards:

```
AREA I    AREA II    AREA III
170       IF(        ( W A G E – 1 . 5 0 ) * ( 4 . 2
          EXTEND     5 – W A G E ) )   1 8 0 , 1 9 0
          EXTEND     , 1 9 0
```

The limit on EXTEND cards is three full EXTEND cards plus at most two characters on a fourth EXTEND card. This is enough to produce the equivalent of a 72-column punched card for all FORTRAN statements. (Note columns 73-80 are ignored by the compiler.)

Warning: In printed output, the compiler lists the set of equivalent eighty-column cards, *not* each MIDITRAN card individually. Thus the three MIDITRAN cards indicated above appear on the program listing with the printed output as *one* printed line, namely the FORTRAN statement shown just before.

Do not confuse the functions of the MIDITRAN "EXTEND" with the FORTRAN "CONTINUATION". "EXTEND" is used to piece together the information on several MIDITRAN cards, so as to create one equivalent eighty-column card. "CONTINUATION" is used to piece together several eighty-column cards, so as to create one FORTRAN statement, too long to fit on a single eighty-column card. As an example of the combined use of "EXTEND" and "CONTINUATION", we show below the six MIDITRAN cards needed for the three lines of FORTRAN program in the middle of page 41:

```
AREA I     AREA II     AREA III
           WRITE(      N P R N T , 1 1 0 )
1 1 0      FORMAT(     6 X , 9 H W A G E   R A T E , 9
           EXTEND      X , 5 H H O U R S , 1 1 X , 5 H
           EXTEND      G R O S S , 1 2 X , 3 H T A X ,
           EXTEND      1 3 X , 3 H P A Y ,
CONT                   / )
```

Note: The statement number has been shortened from 1100 to 110, both in the statement number area of the FORMAT statement and in area III of the WRITE statement.

The ECS system control cards are discussed fully in Chapter XV, section C. There are just three control cards, a *JOB card at the beginning, a *DATA card to indicate the start of the execution phase, and a *END card to indicate the end of the entire job. The *DATA card may be omitted if no actual data are read during the execution phase.

Section C: Data cards.

Data may be read during the execution phase either by means of a standard FORTRAN input statement READ(NREAD, *n) List* or by means of free-format input READ, *List* where, in each case, *List* denotes an input list (see Chapter IV) and *n* is the statement number of a FORMAT statement.

The make-up of suitable data cards for free-format input is discussed in Chapter XV, section B. Here we confine ourselves to standard FORTRAN input, i.e., under FORMAT control.

For data cards to be read during the execution phase, area I of the MIDITRAN card *should be blank.* Area II should be either blank or marked "EXTEND" if needed. The desired information to be read sits entirely in area III, sixteen characters at a time. One equivalent eighty-column data card is made up of one leading MIDITRAN card followed by up to four EXTEND cards. This gives a total of 16 + 4*16 = 80 characters, the exact equivalent of an eighty-column card.

If only the first few columns of an eighty-column data card are actually used, then there is no need to add blank EXTEND cards just to make the number of characters up to 80. Blank characters are understood for the remainder of the equivalent eighty-column card in any case. For example, if only the first twenty columns are actually used, then it suffices to use two MIDITRAN cards, the leading card containing columns 1 to 16 in its area III, the first (and only) EXTEND card having the desired columns 17, 18, 19, and 20 marked in the first four sub-areas of its area III. The absence of further EXTEND cards means that the remainder of the equivalent eighty-column data card is taken automatically to consist of blank characters.

The format specification in the format statement refers to the equivalent eighty-column card, which is the "input record" seen by the computer under MIDITRAN. The "piecing together" of EXTEND cards (if any) and filling in of final blank characters is all done before the format specification starts its work.

CHAPTER XV

I/O UNITS, FREE-FORMAT I/O, AND INPUT DECK STRUCTURE

Section A: Input/Output ("I/O") Devices and Unit Numbers.

The input device is a card reader capable of reading MIDITRAN cards; it has logical unit number 1. The output device is a line printer, which may differ from one installation to another in details, but always has at least 120 character positions per line, and usually 132 character positions per line; its logical unit number is 2. Thus FORTRAN programs should use the statements:

```
NREAD=1
NPRNT=2
```

MIDITRAN does not permit punched-card output from the computer. The maximum record size for unit 1 is eighty characters. The maximum record size for unit 2 is 121 for a line printer with 120 characters to a line, 132 for a printer with 132 characters to a line. It is 132, not 133, in the latter case, because MIDITRAN has an upper limit of 132 characters for *every* logical record.

The first character of an output record destined for the printer is not ignored, as in Basic FORTRAN IV, but is used as a printer control character, i.e., it contains a coded command to the line printer. The code characters and their functions are:

Control character	*Function*
blank	Start the next line (used normally)
0	Skip one line, then start a new line
1	Skip to the start of a new page before printing next line
*	Overprint the previously printed line (avoid this!)

If any other character appears as the first character of the output record, it is treated as if it were a blank in the table above.

Section B: Free-Format Input/Output.

To aid students, MIDITRAN allows for two types of I/O other than the standard FORTRAN variety. One of these, so-called "free-format" I/O, is described in this section; the other, "FORTRAN II" I/O, is described in Chapter XVII, section D.

The concept of an input/output list is defined on page 36, for a simple list, and on pages 117-118, for implied DO loop list items. The *free-format output* statement has the form

PRINT , *List* or PRINT ,

where "PRINT" is one of the keywords in area II of the MIDITRAN card, and the comma following it must be the first non-blank character marked in area III of the card. No unit number is given (the printer is understood), and no FORMAT specification is necessary. The contents of the *list* are printed out, one by one, six to a line, until the list is finished. If there is no list at all (the second alternative above), a blank line is produced on the page, i.e., the command "PRINT," is equivalent to a line skip. In any case, every time a free-format PRINT command is encountered during the execution phase, a new line is started.

Integers and most real numbers are printed out in normal easy-to-read fashion. For very large (more than a thousand million) and very small (less than 0.01) real numbers, the method used is as described and explained on page 8. As an example, suppose the command "PRINT,J,A,B" is reached during the execution phase at a time when memory location J contains the integer 15, memory location A contains the real number -25.27, and memory location B contains the real number −0.00263. Then we get

15 −25.270000 −0.26300000E−02

with exactly eighteen blank characters before the "15", eight blanks after it, and seven blanks between the two real numbers. The effect is that (1) Every number uses up exactly twenty character positions on the printed line; (2) Integers and real numbers in E-format are pushed to the extreme right of the available space, with blank characters preceding them (e.g., 18 blanks preceding the two-digit integer "15"); (3) Real numbers in ordinary form are printed out with eight significant digits, in such a way that the decimal point appears always in the same position (the twelfth position) within the twenty-character area of the line.

In the program on page 6, the initial FORMAT statement can be removed altogether, provided the WRITE statement farther down is replaced by: "PRINT, DEBT, CHARG, PAYMT, REDUC,RMAIN". The output will then be in a more readable form as well. *The same remark holds for all output statements involving output of numbers only.*

There are two similar-looking keywords in area II of the MIDITRAN card (page 10): "READ" and "READ(". The latter, with the left parenthesis, is used for standard FORTRAN formatted input, as described on page 36. The former, "READ" *without* parentheses, is used for non-standard input commands, one of which is the *free-format input statement:*

READ ,*List* or READ ,

where "READ" is the keyword in area II, and the first non-blank character in area III *must* be a comma. If no list is given (second form above) one equivalent eighty-column data card is passed through the card reader without any information being taken from it. If a *List* of memory locations is specified (first form above) numbers are read from a data card, or from data cards, sufficient to fill up all the specified memory locations, in turn.

For free-format input, any markings in *area I of the data cards are ignored.* In area II, the only meaningful mark is "EXTEND", all others are ignored. *We recommend that areas I and II should be left completely blank for free-format input data cards.* The numbers themselves are taken purely from area III. If area III is completely blank, this is taken to mean *one* zero number for the *List.* If more than one number appears in area III, *neighbouring numbers must be separated by commas, but there must not be a comma before the first number in area III, nor after the last number in area III.* Two adjacent commas are allowed, and mean that zero is wanted, for example:

2,0,0,3	can also be written as	2,,,3
0,2,3,0,4,0	can also be written as	,2,3,,4,

Note that initial and final commas are *not* treated as errors, but rather produce (perhaps unwanted!) zero input data numbers.

Conversion of integers to real numbers is done automatically during free-format input. For example, let the command be "READ,I,J,A,B" and let the data card contain "2,0,0,3". Then the contents of I,J,A, and B are set to 2, 0, 0.0, and 3.0, respectively, i.e., the integer 3 is converted to the real number 3.0 automatically.

However, conversion in the other direction (real number to integer) is *not* done during free-format input. The command "READ,K" together with a data card containing the real number 2.3 leads to an error message "MODE ERROR", followed by termination of job execution.

The treatment of *blank characters can be a trap.* Blank characters are *ignored,* completely and utterly, as if they were not present at all. For example, take the commend "READ,K,L" and the data card:

23 45

This does *not* put 23 into K and 45 into L. Rather, the number 2345 is put into memory location K, and the *next* data card is read to obtain an input number for memory location L! Lesson: *Remember those commas between data numbers!*

If there is more than one free-format reading operation during program execution (i.e., if there are several READ statements, or the same READ statement is reached several times during the execution phase), then each reading operation commences on a new MIDITRAN data card, even if the earlier reading operation has not "used up" all the data numbers on the previous data card. For example, suppose "READ,I,J,K,L" is followed by "READ,M,N" and the data cards have (in area III):

Data card 1: 2, −5, 7

Data card 2: 25, 16

Data card 3: 13

Data card 4: 45, 28

The first READ command puts 2 into memory location "I", −5 into "J", 7 into "K"; since the input list has not yet been exhausted, data card 2 is also read, with 25 being put into location "L". The second READ command necessarily starts a new data card, hence 13 is put into location "M", 45 into location "N". The data numbers 16 (on data card 2) and 28 (on data card 4) are not used at all, and are simply "overlooked".

Section C: Input Deck Structure for MIDITRAN.

Every job must start with a "*JOB" card, made up as follows: *Area I must be blank,* area II has a mark in the "*JOB," position, and area III (together with area III of EXTEND cards, if necessary) contains, in sequence: (a) a *job number,* which must be *at least 1000 and at most 9999,* (b) a comma, (c) the letters FTN (for FORTRAN), (d) a second comma, and (e) any additional information desired by your installation, for example, your name, identification of your class or tutorial, etc. An example is:

AREA I	AREA II	AREA III
	* JOB,	1045,FTN,JONES MATHS I

Ask your installation what you should use as your job number. The job number *must* be purely a four-digit number (no letters, decimal point, or other symbols) and it *must not* be preceded by a comma in area III (the comma is already understood as part of the meaning of the mark in area II). It is *essential to leave area I completely blank;* any mark in that area makes ECS refuse to recognize the card as a job card at all, and leads to your cards being "flushed through" the card reader, without any printout from the line printer.

The letters FTN must appear in just that form (*not* "FORTRAN" or "MIDITRAN", but "FTN"). Any misspelling results in the error message "NO SYSTEM OR PROGRAM CALLED . . .", followed by abortion of the job.

The job card is followed by FORTRAN program cards, *main program first,* subprograms (if any) afterwards.

Next comes a "*DATA" job control card, i.e., area II has a mark in the "*DATA" position, the card is blank otherwise. *Do not put any actual data numbers on this control card.* Data cards needed for input during the execution phase, if any, must come *after* the "*DATA" control card. If no input data cards are required by the job, the "*DATA" control card is optional and may be omitted.

Next, if needed, come data cards containing input data to be read. Data cards for free-format input have been described in section B above; data cards for formatted input have been described on page 164.

Finally, the *last card of every job* must be the system control card "*END". This is marked in area II only, in the top two positions on the left side (i.e., marking both "*JOB," and "*DATA" produces an "*END" control card). *Notes:* (1) Do not confuse "*END" with the FORTRAN "END" statement. The FORTRAN "END" indicates the end of one program unit; whereas the "*END" is an ECS system control card, indicating the finish of an entire job. (2) The "*END" control card can be omitted, and the job will run nonetheless; however, this is a dangerous thing to do, and we recommend strongly against it, for the following reason: If the next job, following yours in the queue, has a faulty *JOB card, and your job has no *END card at its end, then you are likely to get utterly unintelligible diagnostic messages on *your* job output. The effect of the *END card as the last card of your job is to tell the ECS system loudly and clearly: "Here is the finish of my job; do not bother me with error messages arising from someone else's errors." An additional reason for including the *END card is to get your output out of the printer if there is no other student job immediately after yours. The *END card makes the printer move the paper forward so you can tear if off to get your output. Without the *END card, you may just stand there waiting with part of your output still inaccessible inside the printer.

CHAPTER XVI

THE PRINTED OUTPUT. DIAGNOSTIC MESSAGES.

Section A. Appearance of the Printed Output.

We start with an example of a correct program, namely the program on page 6.

```
MONASH UNIVERSITY EDUCATIONAL COMPUTER SYSTEM      DATE 17/09/74      TIME 17H 12M

*JOB,1234,FTN,BLATT

 *001      1 FORMAT(7E16.6)
 0002        DEBT=300.00
 0003        PAYMT=22.40
 0004        CHARG=0.008*DEBT
 0005        REDUC=PAYMT-CHARG
 0006        RMAIN=DEBT-REDUC
 *007        WRITE(2,1)DEBT,CHARG,PAYMT,REDUC,RMAIN
 *008        STOP
 *009        END

TOTAL ERRORS  =  0000

TIME    =     5

    0.300000E 03     0.240000E 01     0.224000E 02     0.200000E 02     0.280000E 03

STOP    =     0

RUN PROFILE

   LINE       2    3    4    5    6    7    8
   EXEC       1    1    1    1    1    1    1

TOTAL JOB TIME =      7 SECONDS
```

Let us go through this output, line by line, explaining as we go along. The top line gives the date (in the order "day/month/year", *not* "month/day/year"!) and time of day (hours and minutes, hours up to 24); then comes a listing of the *JOB card, followed by the listing of the FORTRAN program, with line numbers assigned by the MIDITRAN compiler at the left. These line numbers have nothing to do with FORTRAN statement numbers; the line numbers merely identify each line uniquely. The initial asterisk in *001 for the first program line means that MIDITRAN cards (rather than eighty-column punched cards) were used, and the keyword "FORMAT(" was taken from area II of a MIDITRAN card. In line 0002, area II of the MIDITRAN card was blank, as it always is for an arithmetic assignment statement. In lines *007, *008, and *009, the keywords "WRITE(", "STOP", and "END", came from markings in area II.

The next two messages, from the MIDITRAN compiler, tell us that the compiler detected no errors during compilation, and the time needed for the compilation phase was 5 seconds (this time is mainly governed by the speed of the line printer, not by internal working of the computer). The output thereafter comes from the execution phase (note the *DATA control card is *not* listed on the output, even though it was present).

STOP = 0 means that the program came to a halt on a FORTRAN statement STOP. The FORTRAN language permits the form "STOP *n*" rather than just "STOP", where *n* is a number of up to four digits, each digit being in the range 0 to 7, inclusive. If we had written "STOP 225" in line *008, the execution time output should have terminated with "STOP = 225" instead of "STOP = 0".

The subsequent "run profile" lists the line number of each executable FORTRAN statement which was actually reached in execution (thus, lines 1 and 9 are not included), and, below it, the *number of times* this statement was reached. In our case, execution proceeded straight down the program, starting with line 2 and ending at line 8, each statement being reached just once. But if there are loops in the program (e.g., see page 31), then statements within a loop are reached more than once, and the run profile information can be very helpful. (Note: Due to computer memory limitations, the run profile is incomplete for very long FORTRAN programs; but it is still useful).

The final message tells you the total time for the job (usually, governed by the speed of the line printer).

So much for a correct program. Next, we give the output from the horribly incorrect program on page 47.

```
MONASH UNIVERSITY EDUCATIONAL COMPUTER SYSTEM     DATE 17/09/74     TIME 17H 13M

*JOB,3456,FTN,BLATT

     C      SAMPLE PROGRAM WITH DIAGNOSTICS
 *001       READ(1,1  A,B,C
ERROR NO. 09
 *002       FORMAT(4F10 0)
ERROR NO. 58
ERROR NO. 46
 *003       IF(N)3,3
ERROR NO. 16
 *004       GO TO77
ERROR NO. 52
 0005       A=A*B/C**2
ERROR NO. 52
 *006       WRITE(2,77)
ERROR NO. 62
 *007   77  FORMAT (17H  DATA  UNACCEPTABLE)
ERROR NO. 55
ERROR NO. 47
 *008       STOP
ERROR NO. 11
UNDEFINED ST NO 1
UNDEFINED ST NO 3

TOTAL ERRORS  = 0011

TOTAL JOB TIME =    8 SECONDS
```

The errors are identified by numbers only (in some versions of MIDITRAN, short messages appear after the numbers). The actual diagnostics are listed in Section B of this Chapter. Error no. 9 is "WRONG CHARACTER"; the wrong character is the "A", as explained on pages 48-49. Unfortunately, MIDITRAN does *not* inform you just where the "wrong character" is within the statement. Error no. 58 is "FORMAT WITHOUT STATEMENT NUMBER", plain enough. Error no. 46 is "INVALID FORMAT ITEM", and comes from the missing decimal point in "4F10 0". Error no. 16 is "COMMA EXPECTED", as it is after the second "3". Error no. 52 is "UNREACHABLE STATEMENT", i.e. missing statement number on a statement following an arithmetic IF (line 4) or following a GO TO (line 5). Error no. 62 is "FORMAT NUMBER NEEDED", see the explanation on page 49. Error no. 55 means "EXECUTABLE STATEMENT EXPECTED", because of the "GO TO 77" in line 4. Error no. 47, "INVALID DUPLICATION FACTOR", refers to the 17 in "17H": the *count* is wrong, since it fails to include the final letter "E" of the subsequent character string.

The remaining three diagnostic messages, issued at the end of the compilation attempt, tell us that the END card is missing (error no. 11), and that the logic of the program demands statements with statement number 1 (from line 1) and 3 (from line 3), but no statements with these statement numbers are present.

With the MIDITRAN compiler under the ECS system, the presence of errors during the compilation phase prevents any attempt at execution of the job. Thus, we are told the total number of errors found, the total job time, and sent packing.

*If nothing at all is printed out, your *JOB card was not recognised as such, and it is at fault.*

Section B. Compile Time Error Messages from MIDITRAN.

Most of the error messages from the MIDITRAN compiler are associated with error numbers, e.g., "ERROR NO. 58". The first list below contains all these numbers, with the appropriate message following the number. In some, but not all, versions of MIDITRAN the short message (in capital letters) also appears on the printed output, not just the number of the error. In most cases, the brief message is amplified by a more detailed explanation of what is likely to be wrong and/or some suggestions how to fix it.

*A very few compile time error messages are unnumbered; these are given at the end of this section. Other unnumbered messages are issued during execution time; they are explained in the next section. No output at all indicates that the *JOB card is faulty (marks in area I, or insufficient mark in area II).*

LIST OF NUMBERED DIAGNOSTIC MESSAGES

1 JOB TOO BIG. The compiler has run out of computer memory. Try dividing large program units into smaller subprograms.

3 ILLEGAL CONTINUATION CARD. Have you tried to use more than three continuation cards? Or have you mismarked area I of a card accidentally?

4 STATEMENT NUMBER ON CONTINUATION CARD. Not permitted.

5 INVALID CHARACTER FOR AREA I. Area I must contain either a statement number, or a marking "comment", or a marking "cont", or remain blank. Have you marked two positions in the same vertical column? (If ordinary punched cards are being used, errors 3,4, and 5 often indicate a mispunched FORTRAN statement starting in column 1 of the card instead of column 7).

6 DUPLICATE STATEMENT NUMBER. The statement number on this statement duplicates a statement number used earlier in the same program unit.

7 EXPECTED IDENTIFIER IS ILLEGAL. The compiler expected an identifier name at some point in this statement; instead, it found either an illegal name (more than six characters) or something which is not a name at all (leading character not alphabetic).

8 INVALID DIMENSION. You must not (a) Specify more than three dimensions for one array, (b) Specify an array with a total number of elements in excess of 4096, (c) Specify an array with variable dimensions (see Chapter XVII, section D) in a main program, (d) Specify an array with variable dimensions unless the array name as well as the names of the variable dimensions are themselves dummy variables in the argument list of the subprogram.

9 WRONG CHARACTER. Either (a) Something which is not in the FORTRAN character set at all (wrongly marked sub-area of area III), or (b) A proper FORTRAN character in an improper place, for example the "$" in "READ(1,150$".

10 MISSING SUBPROGRAM HEADER. Either (a) You have forgotten the *DATA card and the compiler is now trying to translate your first data card, or (b) You have put a FORTRAN statement END somewhere in the middle of your program, or (c) You have failed to put your main program at the *beginning* of the job deck, so the compiler now thinks of your main program as a "subprogram without a header", or (d) You have mismarked a FUNCTION or SUBROUTINE header card badly enough so the compiler did not recognize it as such.

11 END CARD MISSING. Either (a) You have forgotten to put END just before the *DATA control card, or (b) The first program unit in your deck is a subprogram rather than the main program, or (c) A SUBROUTINE or FUNCTION statement appears in the middle of another program unit, or (d) You have omitted the END statement for the immediately preceding program unit.

12 UNIDENTIFIABLE STATEMENT. The compiler is baffled. A common cause is that a FORTRAN keyword such as GO TO has been written out in area III of the card, instead of using the marking position for that keyword in area II of the card.

14 ILLEGAL SEQUENCE. See page 151, bottom,for the correct sequence of FORTRAN statements within one program unit; then shuffle your cards accordingly.

15 STATEMENT NUMBER EXPECTED. The compiler expected a statement number, found something which is not a number, for example "GO TO HADES" instead of "GO TO 666". A common cause is omission of a comma in free-format READ or PRINT statements. "PRINT A,B,C" causes this diagnostic because the compiler expects *either* "PRINT,A,B,C" for a free-format print statement, *or*, say, "PRINT 120,B,C" where 120 is the statement number of a FORMAT statement, for a FORTRAN II type input/output statement (see Chapter XVII, section D). Thus if the keyword PRINT is not followed by a comma, then it must be followed by a statement number, which is not true for "PRINT A,B,C".

16 COMMA EXPECTED. The compiler expected a comma, did not find it. Check page 151 for correct syntax of FORTRAN statements.

18 INVALID STATEMENT FOR LOGICAL IF. In the form IF (*condition*)*statement*, described on page 178, the *statement* is either non-executable, or is a DO statement, or is another logical IF statement.

19 WRONG = SIGN. The equality sign within a DO statement is incorrectly placed, or is missing altogether. See page 85, particularly the *Note*.

20 INVALID DO PARAMETER. One of *k, l, m* in the form DO *n J=k,l,m* is neither an unsigned integer constant nor a simple (non-array) integer variable name.

21 DOUBLY DEFINED NAME. The same identifier name appears twice in the COMMON list, or it appears twice in DIMENSION statements.

22 SUBPROGRAM INCONSISTENT. This attempt to define a subprogram, or this attempt to invoke a subprogram, is inconsistent with an earlier appearance of the same subprogram name. Examples: (a) CALL SLAVE(X) followed later by a FUNCTION type subprogram header FUNCTION SLAVE(X), or (b) followed by a header with the wrong number of dummy arguments like SUBROUTINE SLAVE(A,B), or (c) either of the above in reverse order.

23 EXCESSIVE NESTING OF DO LOOPS. One DO may be inside another, and that inside yet another, etc., but not more than ten deep. Remember that implied DO loop items in input/output lists are counted as part of the "nest".

24 WRONG TERMINATION OF A DO LOOP. The statement in question is the final statement of a DO loop (its "termination"), and as such violates the rules: it must be an *executable* statement, *other than* one of: GO TO, IF, RETURN, STOP, PAUSE, DO.

25 DO LOOP VARIABLE ALREADY IN USE IN OUTER LOOP. Nested DO loops each require their own unique loop variable, i.e., "DO 120 J=1,5" must not appear *within the range* of "DO 500 J=3,9", since the same memory location J is trying to do two jobs at once. Note that implied DO loop list items in input/output lists are treated as DO loops for the purpose of this diagnostic message.

26 CONSTANT ILLEGAL IN INPUT/OUTPUT LIST. "PRINT,A,B" is legal but "PRINT,11.5,23.7" is illegal. See page 36.

27 SUBPROGRAM NAME ILLEGAL IN INPUT/OUTPUT LIST. A function or subroutine name must not be a list item.(One exception: If this is itself a FUNCTION subprogram, e.g., FUNCTION FUNC(X), then FUNC may appear in an I/O list within *this* program unit, only; no other function name may appear, and no subroutine name at all).

28 UNMATCHED BRACKETS IN IMPLIED DO LOOP. The brackets enclosing an implied DO loop list item do not match up properly.

29 STATEMENT NUMBER IN THIS DO HAS OCCURRED EARLIER. In DO *n J=k,l,m* the statement with statement number *n* is the termination statement of the loop, and must come *later than* the DO statement, within this same program unit. Here, a statement with number *n* has already occurred earlier. Have you used duplicate statement numbers in your program?

31 MISSING COMMAS IN IMPLIED DO. The rules for implied DO loop list items require a comma just before the name of the loop variable; see page 117 for the correct form.

34 EXCESSIVE SIZE OF COMMON. In MIDITRAN, the first program unit which specifies COMMON storage at all, also decides the total amount of COMMON storage reserved by the compiler. If a later program unit asks for more COMMON storage than that, this diagnostic is the result. *Remedy:* Specify COMMON JUNK(. . .) in the *main* program (which must come first in your job deck), with the number . . . chosen as big as the biggest amount of COMMON used in any of the subprograms. (Or, if the main program itself wants to use some COMMON storage, follow the COMMON declaration which is really wanted with another declaration COMMON JUNK(. . .) to bring the total size of COMMON up to the right amount).

35 ARITHMETIC EXPRESSION STARTS INCORRECTLY. The compiler expects an arithmetic expression; such an expression must start with one of: A valid variable name, a plus sign +, a minus sign -, or a left parenthesis (. Anything else at the beginning of an expression or sub-expression is illegal. Note that an *invalid* variable name (more than six characters) can cause this error.

36 OPERATOR EXPECTED. Either (a) An operator is missing, like "(A+B)C" instead of "(A+B)*C", or (b) A variable name has more than six characters (the compiler expects the seventh character to be an operator like * or +).

37 BRACKETS DO NOT MATCH. See the second *note* on page 19.

38 LEFT HAND SIDE OF = ILLEGAL. In an arithmetic assignment statement: *NAME=expression* the *NAME* is not a valid name of a memory location. Usual cause: Omission of the DIMENSION declaration for an array name, or rejection of that DIMENSION statement by the compiler because of some error. For example, if the name MARK has not appeared in a DIMENSION statement, or has appeared in an erroneous DIMENSION statement, then "MARK(J)=L" causes error no. 38. *Note* that

the general discussion of this type of error, on page 107, must be modified for MIDITRAN, since MIDITRAN does not recognize arithmetic statement functions. Thus, "MARK(J)=L" is *not* interpreted as an attempt to define an arithmetic statement function. Conversely, another possible cause for error 38 is a perfectly valid (in standard FORTRAN) definition of an arithmetic statement function. If this is your trouble, replace the arithmetic statement function by a subprogram of FUNCTION type, see Chapter XII.

39 MIXED MODE ERROR. See page 17.

41 DUMMY VARIABLE APPEARS IN A COMMON OR DATA DECLARATION. In FUNCTION OUCH(OOF) the dummy variable name OOF must not be placed into COMMON storage, nor initialized by a DATA statement (see Chapter XVII, section D for the latter).

42 INVALID SUBSCRIPT. *Either* (a) One of your subscripts is not a valid subscript expression according to the rules on page 100; *or* (b) You have specified too many, or too few, subscripts for an array (i.e., the number of subscripts disagrees with the corresponding number in the DIMENSION declaration for this array name). Have you attempted to use an array name without any subscripts at all?

43 NO RETURN. Each FUNCTION or SUBROUTINE subprogram must contain at least one RETURN statement; the compiler has encountered the END statement already, but no RETURN statement. This may be a secondary diagnostic, arising from rejection of an actual RETURN statement for some other reason.

44 IMPROPER USE OF A SUBROUTINE NAME. *Either* (a) One name is used both as a subroutine name and as the name of a simple variable, in the same program unit; *or* (b) A subprogram name has been used both as if it were the name of a FUNCTION subprogram and the name of a SUBROUTINE subprogram; e.g., the statements "Y=OUCH(X)" and "CALL OUCH(X)" must not appear within one and the same program unit.

45 INVALID EXTERNAL DECLARATION. See Chapter XVII, section D; only subprogram names can be declared to be EXTERNAL, and the current subprogram cannot be declared EXTERNAL within itself.

46 INVALID FORMAT ITEM. Some item within this FORMAT statement is in error. See page 79 for a summary of the rules.

47 INVALID DUPLICATION FACTOR. The duplication factor *n* in a field descriptor (for example, in *n*I*w* or *n*X) is invalid; in MIDITRAN *n* must not exceed 132. Did you miscount the length of an *n*H. . . string?

48 FUNCTION WITHOUT DUMMY ARGUMENT. Every FUNCTION subprogram must have at least one dummy argument. This one doesn't.

49 INVALID CALL STATEMENT. In "CALL HELP(X)" the name "HELP" must be the name of a SUBROUTINE subprogram, *not* of a FUNCTION subprogram. This is essentially the same as diagnostic 44(b); you get error 44 if "CALL OUCH(X)" precedes "Y=OUCH(X)", and error 49 if "Y=OUCH(X)" comes first. The *first* invocation of a subprogram decides what type of subprogram is expected as far as the compiler is concerned.

50 INVALID RETURN. The RETURN statement is not permitted in the main program; use STOP instead.

51 INVALID USE OF FORMAT LABEL. This label (statement number) has appeared earlier as the statement number of a FORMAT statement. You have *either* (a) used this same number as the statement number of the current statement (in that case error no. 6 will also appear), *or* (b) you have used it as if it were the statement number of an executable statement, for example "GO TO 120" where 120 is the number of an earlier FORMAT statement.

52 UNREACHABLE STATEMENT. This statement has no statement number but follows immediately after one of: GO TO, STOP, RETURN, arithmetic IF. This statement can therefore never be reached in execution.

53 END OF STATEMENT EXPECTED. Non-blank characters have been detected following the logical conclusion of a statement, for example the final period in "GO TO 120." causes this error.

54 STATEMENT NUMBER ON DECLARATIVE STATEMENT. In MIDITRAN, the declarative statements FUNCTION, SUBROUTINE, DIMENSION, COMMON, and END *must not* be given statement numbers.

55 EXECUTABLE STATEMENT EXPECTED. The statement number *n* of the current statement has been used earlier in the program as if it labelled an executable statement (for example, "GO TO *n*" has appeared earlier); yet now *n* is being used for a FORMAT statement, (This is essentially the same error as 51b; whether you get error 51 or 55 depends on whether "GO TO *n*" or some such statement comes *before* or *after* the FORMAT statement numbered *n*).

56 REPEATED DUMMY VARIABLE. The same name appears twice in the list of dummy variables in a subprogram header statement.

57 CURRENT FUNCTION NAME IN TYPE DECLARATION. If you want IDIOT(X) to return a real, rather than an integer, function value, you *must not* write "REAL IDIOT" *inside* the function subprogram; rather, the header must be "REAL FUNCTION IDIOT(X)". See the warnings against explicit type declarations of any sort in Chapter XVII, section D. Use "FUNCTION DUNCE(X)" instead!

58 FORMAT WITHOUT STATEMENT NUMBER. Illegal.

59 TOO MANY CONTINUATION OR EXTEND CARDS. No more than four EXTEND cards, no more than three CONTINUATION cards, are permitted in MIDITRAN. Usually accompanied by error message 3 as well.

60 SUBPROGRAM INVOKES ITSELF. Illegal. Don't "CALL YELP" *inside* SUBROUTINE YELP.

61 TOO MANY SUBPROGRAMS. No more than ten in MIDITRAN. Combine two or more subprograms into one larger subprogram.

62 FORMAT NUMBER NEEDED. In "READ(1,*n*)*List*" or "WRITE(2,*n*)*List*", statement number *n* was used earlier in the program for a statement which was not a FORMAT statement, or has appeared within a statement such as "GO TO *n*".

63 ILLEGAL CONSTANT. A constant is too large for the computer, or otherwise invalid. See p. 175.

UNNUMBERED ERROR MESSAGES

DATA ERROR IN xxx. Line number xxx contains an erroneous DATA declaration; see Chapter XVII, section D for the rules. Only the *first* erroneous DATA declaration is diagnosed by the compiler; later errors are overlooked.

DO OVERLAPS AT xxx. The DO loop starting in line number xxx overlaps the DO loop terminated by the current statement.

NON-TERM DO AT xxx. The DO loop starting in line number xxx is not terminated before the END statement (for example, "DO 250 K=2,5" without any statement number 250).

UNDEFINED ST NO xxx. The statement number xxx is used within the program unit (for example, as a DO termination or in "GO TO xxx") but there is no statement numbered xxx prior to the END statement.

Section C: Execution Time Error Messages.

A number of errors can be, and are, caught only after the program has gone into execution; and even then, not all errors are caught. If your program compiles without error messages, and then executes without error messages, it can still give wrong answers. Even worse, it can give right answers under one system (for example, ECS) and wrong answers under another system! Two likely causes of this are:

1. At the start of the execution phase, MIDITRAN sets to zero the contents of all memory locations named in the program. Thus, if "PRINT,K" is the very first statement and K has not been set at all, zero will be printed out. This is *not* a general rule for all FORTRAN systems, and may cause a program to work correctly in MIDITRAN, but fail (because of garbage in K) on another computer.

2. Zero raised to the power zero is not defined mathematically. In MIDITRAN, the command "K=L**M", reached when L and M both contain 0, produces 0 in location K. Corresponding statements hold for "A=Z**M and "A=Z**ZZ" at a time when Z and ZZ contain 0.0.

However, quite a few errors *are* detected. The typical appearance of an execution time error message is:

–VE ARG IN SQRT

ERROR IS IN LINE 2

This tells you that your program died in an attempt to take the square root of a negative number, while obeying the statement in line 0002 of the MIDITRAN listing (2 is the *line number* to the extreme left, *not* a FORTRAN statement number).

*We remind you that no printout whatever, with the whole job deck just flushed through the card reader, indicates that the computer did not recognize your *JOB card at all as a job card;* this may be due to some extra mark in area I, or to an insufficient mark (overlooked by the card reader) in the right place in area II.

LIST OF EXECUTION TIME ERROR MESSAGES

–VE ARG IN SQRT. Attempt to take the square root of a negative number.

–VE ARG IN ALOG. Either ALOG(X) or X**Y, with X negative at this moment. See page 63.

0 ARG IN ALOG. Either ALOG(X) or X**Y, with X zero at this moment. See page 63.

DIV BY 0. An attempt to divide by zero.

ARITH OVFL. Arithmetic overflow. A number has become too large for the computer to handle properly. See Chapter XVII, section A, for the largest numbers permitted. Some of the possible causes are: "J=(K*L)/M" with K*L too big (even though the final J would have been all right!); IFIX(X) with X larger than the largest integer; zero raised to a negative power; SIN(X) or COS(X) or EXP(X) with X too large for comfort; an excessively large input number on a data card; etc.

STK OVFL. An arithmetic expression proved too complicated to evaluate. Break it up into simpler sub-expressions.

BOUNDS ERROR IN xxx
SUB = nnn
SUB = mmm
An array element of array xxx was called for, with subscripts of value nnn (first subscript) and mmm (second subscript). One or more of these subscript values is zero, or negative, or too large for the declared DIMENSION of array xxx. All subscripts are listed, in sequence.

GOTO() nnn. In a computed GO TO (. . .),*J* the value of *J* was nnn. This value is either zero, or negative, or larger than the number of labels in the bracketed list (. . .).

ILLEGAL DO PAR
xxx yyy zzz
This message is given when the CONTINUE statement is reached for a DO loop "DO *n J=K,L,M*" with one or more of *K, L, M* illegal. The three numbers in the second line are the values of *K, L,* and *M*, respectively. One or more of them is negative or zero. Note the loop has already been traversed once at this moment.

UNDEF SUBPROG xxx. A subprogram with name xxx was expected, could not be found. See page 107.

ILLEG PARAM – xxx. *Either* (a) An actual argument for subprogram xxx is of incorrect type (integer rather than real, or vice versa), *or* (b) Subprogram xxx could not be found. *Note:* xxx is the name of the subprogram, *not* of the faulty parameter.

RE-ENTRANT CALL TO xxx. An attempt to call a subprogram from itself, in an indirect fashion. For example, SUBA calls SUBB, and then SUBB in turn calls SUBA.

ILLEGAL UNIT. The logical unit number must be 1 for a READ, 2 for a WRITE.

REC OVFL. Attempt to write a record of total length more than 132.

ILLEG CHAR. A character on a data card was either not a FORTRAN character, or was invalid in its position (e.g., an alphabetic character in a numeric field).

MODE ERROR. (Formatted I/O) An inconsistency between the mode (integer or real) of an item in the input/output list, and the corresponding field descriptor in the FORMAT statement (e.g., I10 for the list item X); *or* (Free-format READ) The READ statement asked for an integer but encountered a real number (with decimal point) on the data card.

ILLEG FORMAT. An error, not detected during compilation, has been found in the FORMAT statement used by the READ or WRITE statement now being executed.

INSUFFICIENT DATA. You ran out of data cards. If deliberate, it is poor programming technique.

TIME LIMIT. You have exceeded the limit on execution time for student jobs at your installation (compilation time is not counted in this).

PAGE LIMIT. You have exceeded the limit on printed output pages at your installation.

Occasionally, you may get the termination message:

STOP = 0
ERROR IS IN LINE xxx

This means that line number xxx contains an END statement which should not have been, but was, reached during program execution. There should have been a STOP statement before it.

The following messages come from *faulty *JOB cards.*

ILLEGAL JOB NUMBER – nnn. The job number *n* in "*JOB,*n*,FTN,*name*" is not numeric, or less than 1000, or greater than 9999. Did you perhaps mark a comma within area III before the job number? If so, the card is read as "JOB,,*n*,FTN,*name*" with an extra comma, and the job number is taken to be an (illegal) zero.

NO SYSTEM OR PROGRAM CALLED xxx. You have misspelled "FTN" in "*JOB,*n*,FTN,*name*".

PROCEDURE REQUESTED ILLEGAL. Same error as above.

In all these cases, the rest of your cards, following the job card, are not listed on the printed output, they are just flushed through the card reader.

CHAPTER XVII

RESTRICTIONS AND EXTENSIONS IN MIDITRAN

Section A: Word Length and Precision.

Within any computer, there are limits on the size of the numbers that can be stored (see page 59), and for real numbers, there is only a limited number of significant digits. These limits differ from one machine to another. Since the MIDITRAN compiler has been implemented on a number of different machines, the table below contains the limits for those machines. If your computer is not within that list, you must ask your installation what the limits are for your computer.

The table starts with the most negative integer and the most positive integer permitted on each computer. Any attempt to calculate an integer outside these limits causes the "ARITH OVFL" (arithmetic overflow) error message, and termination of job execution. Next is information about real numbers in the computer; here the sign does not matter; we are concerned with the absolute magnitude of the number and with the number of significant digits. The minimum and maximum magnitudes are only approximate (within the computer, one works with powers of 2, not with powers of 10), and the same holds for the number of significant digits. Any real number of magnitude less than the "minimum" is replaced by zero within the computer; for example, 3.26E-85 is stored (approximately) as such if your computer is a Control Data, but is treated as if it were exactly zero in all the other computers in the Table. Real numbers of magnitude greater than the "maximum" produce arithmetic overflow. For a discussion of precision, significant digits, and their consequences, see pages 60–62.

The final column of the Table refers to a full FORTRAN, rather than Basic FORTRAN, feature of MIDITRAN, namely the fact that strings of characters can be read into computer memory, and printed out again, by means of the A-format code specification (see section D). The longest character string which fits within one memory location is 4 characters in most machines, but 6 characters in a Burroughs B5500.

MACHINE LIMITATIONS OF MIDITRAN ON VARIOUS COMPUTERS

COMPUTER	INTEGERS		REAL NUMBERS			MAXIMUM
	MINIMUM	MAXIMUM	MINIMUM	MAXIMUM	SIGNIFICANT DIGITS	CHARACTERS PER WORD (A-FORMAT)
DEC PDP–11	–2147483648	+2147483647	10**(–38)	10**38	8	4
Burroughs B-5500	–549755813887	+549755813887	10**(–47)	10**68	11	6
Control Data 3200, 3300, 3500	–8388607	+8388607	10**(–308)	10**308	11	4
IBM-360	–2147483648	+2147483647	10**(–78)	10**75	7	4
ICL-1900	–8388607	+8388607	10**(–76)	10**76	11	4

Section B: FORTRAN Language Restrictions and Limitations in MIDITRAN.

Every FORTRAN statement except arithmetic assignment statements starts with a *keyword*, for example, IF, READ, DO, etc. Thus a compiler recognizes what sort of statement it is by looking at the first word and comparing it with all the keywords. The MIDITRAN compiler simplifies this process, by demanding that a keyword, if present as the first word, must be marked in area II of the MIDITRAN card. Thus, "GO TO" in "GO TO 250" is recognized by MIDITRAN if the "GO TO" position in area II is used; it is not recognized by MIDITRAN if the letters "GO TO" are spelled out in area III; rather, MIDITRAN then issues diagnostic message 12: "Unidentifiable statement."

In Basic FORTRAN, the keywords occur, if at all, *only* at the beginning of a statement (this is part of the difference between Basic FORTRAN and full FORTRAN). In MIDITRAN, this is not quite true, since MIDITRAN includes some features of full FORTRAN; see section D for these situations.

There is *no EQUIVALENCE* statement in MIDITRAN; *no arithmetic statement functions* exist; and at present there are *no sequential file operations* in the system, thereby *eliminating* the FORTRAN statements: REWIND *n*, BACKSPACE *n*, END FILE *n*, READ(*n*)*List*, and WRITE(*n*)*List*. None of these will work. (Future versions of MIDITRAN may allow for sequential file operations, so ask at your installation if in doubt).

Arrays must not have more than 4096 separate array elements, e.g. DIMENSION MARK(10,500) causes error message 8, "illegal dimension", because 10*500=5000 exceeds 4096.

In format codes E*w.d*, F*w.d*, and I*w*, the constant *w* must not exceed 64, and *d* must not exceed 32. In format codes *n*H.. and *n*X, *n* must not exceed 132. The total length of every I/O record (see page 68 ff) must not exceed 132.

DO loops may be nested inside each other, see page 89, but the nesting must not be more than ten deep. If, within some outer loop, there is an I/O command with implied DO loop list items in the list (page 117), then such implied DO loops are counted within this nesting. However, student programs are not likely to run up against this limit.

The PAUSE and PAUSE *n* statements exist, but they do not stop the computer; they act like CONTINUE statements.

No more than *three* continuation cards are permitted (Basic FORTRAN permits up to five). The limits on EXTEND cards have been stated in Chapter XIV.

Non-executable statements (see the list on page 151), except for FORMAT, *must not* be given statement numbers; FORMAT statements *must* be given statement numbers.

The number of subprograms in a single job must not exceed ten. If subprograms are used, the main program must come first in the job deck, immediately after the *JOB card. If COMMON storage is used, the first program unit with a COMMON declaration in it, encountered by the compiler during the compile phase, defines the total number of words allotted to the COMMON storage area. Program units encountered thereafter must not demand more total COMMON area than that. (You can always get around this by making the main program demand as much COMMON storage as the hungriest subprogram, even if the main program does not itself use all of the storage area.) Naturally, the total memory size of the computer is limited, and this may make your job "too big" to compile, or to execute, or both. Beginners are not likely to encounter this problem.

At the beginning of the execution phase, the contents of all named memory locations are set to zero. Hence, *failure to place a number into location A prior to using the contents of A will not be detected by the system.*

If I and J contain 0, and A and B contain 0.0, the values of I**J, A**J, and A**B are all taken to be zero, rather than treated as errors. They may be treated as errors by other FORTRAN systems.

Section C: Additional Built-in Functions in MIDITRAN.

Besides the built-in functions listed on page 95, MIDITRAN provides two more, called RANF(X) and PLOT(X), respectively.

The statement Y=RANF(X) puts a "random number" into memory location Y; this random number is a real number, in the range 0.0 to 1.0. The numerical value of X is not used at all, and is of no consequence. Note that Y=RANF(A)+RANF(B) puts the sum of two different random numbers into memory location Y. We *warn against* trying to do this by Y=RANF(A)+RANF(A); this will work with MIDITRAN, but will fail with an "optimising" FORTRAN compiler, since such a compiler will recognize that this statement is apparently equivalent to Y=2.0*RANF(A), and will compile it as such – which is not at all what the programmer wanted! "Optimization" can lead to peculiar results!

The statement Y=PLOT(X), in spite of appearances, is an *output* statement, producing one printed line on the line printer. If the value of X is *positive,* this printed line contains only one character, an asterisk "*", in one of a possible 120 print positions along the line; the actual position is determined by the value of X. For example, if X equals 4.0, the asterisk appears in the fourth position (i.e., the line consists of 3 blank

spaces, then an asterisk, then blanks to the end of the line). The same asterisk appears if X equals 3.7 or 4.4, say, i.e., the value of X is rounded to the nearest integer. If X is greater than 120.0, the asterisk appears in position 120 (extreme right); if X lies between 0.0 and 0.5, the asterisk appears in position 1 (extreme left). If the value of X is *negative*, the printed line starts with a number of adjacent asterisks, until the position is reached where the asterisk appears for the corresponding positive value of X. Thereafter, the printed line is blank. Thus negative values of X are used to produce a "histogram" effect. The statement Y= PLOT(X) does not alter the content of memory location Y.

Section D: Extensions of MIDITRAN beyond USA Standard Basic FORTRAN IV.

In this last section we summarize, and explain briefly, the additional features of MIDITRAN which go beyond USA Standard Basic FORTRAN IV.

1. *Character Set:* Additional special character "$".
2. *Constants:* Additional type, called *Hollerith constant,* is a string of adjacent characters which may be stored into a named memory location. The three-character string "BUG" may be stored into memory location MEMO by the command: "MEMO=3HBUG". The form of the Hollerith constant to the right of the equality sign is *n*H. . . where *n* is the number of characters in the string, and the string itself follows the letter H immediately. Blank characters are counted. The maximum value of *n* is quite small, either 4 or 6, see the Table in section A, last column. If you want to store more characters than that you must break up your character string into short sub-strings, storing each in a separate memory location. The memory location, or locations, must be of a type integer.
3. *Identifier Names:* Up to *six* characters are permitted.
4. *Arrays:* Three-subscript arrays are permitted; avoid them. Array names may be declared in ways other than by a DIMENSION statement (see later).
5. *Arithmetic Operations:* A**B**C is permitted, and is evaluated as (A**B)**C. Avoid this like the plague! The same expression is evaluated as A**(B**C) in IBM FORTRAN, for example.
6. *Additional Statements and Options:*

a) *Explicit type declaration* statements can be used to override the convention concerning the first letter of an identifier name. REAL IDIOT declares IDIOT to be the name of a real variable, in spite of the starting letter "I". Similarly, INTEGER FOOL declares FOOL to be the name of an integer variable, in spite of the starting letter "F". Avoid doing this!!

b) These explicit type declarations, as well as the COMMON statement, are permitted to *declare array names* with the array dimensions. For example, COMMON A,B(5),C does in one statement the work of two Basic FORTRAN statements, namely:
DIMENSION B(5)
COMMON A,B,C
Similarly, REAL LOON(10) declares LOON to be a real variable vector array of dimension 10.

c) The *DATA* declaration is used to set initial values for particular variables, to apply at the very start of the execution phase. For example:
DATA NPRNT/2/,ALPHA,BETA,GAMMA/4.78,1.2E+4,0.0092/
has the effect that, *at the start of the execution phase,* memory location NPRNT contains the value 2, ALPHA the value 4.78, BETA 12000.00, and GAMMA 0.0092. These values are starting values only, and may be altered during the execution phase, for instance by execution of a statement such as BETA=BETA+1.5. The DATA declaration must not be used for variables in COMMON storage. Thus, if NPRNT has been placed into COMMON, then the above DATA declaration is invalid, and we must come back to the simpler statement: NPRNT=2.

d) The following *additional things are possible in formatted input/output:*
i) The carriage control character is effective, see page 165.
ii) Nested brackets are allowed, e.g., FORMAT (2(I6,4(I3,I2))).
iii) Character strings (Hollerith constants, see above) may be read and/or written. The format descriptor is *n*A*w*, where *w* is the length of each string, and *n* is the usual repetition factor. The sequence of commands

```
    READ(1,250) J,K,L
250 FORMAT(A4,2A3)
    WRITE(2,250) J,K,L
    WRITE(2,250) K,J,L
```

together with a data card containing the word "ELIGIBILITY" has the following effect: The character string "ELIG" is placed into memory location J, "IBI " into memory location K (note the blank character at the right end of the string, as a filler for a four-character memory location), and "LIT " into memory location L; the final character "Y" is ignored. The output from the two WRITE commands consists of the two lines:

```
LIGIBILIT
BI ELILIT
```

In each case, the initial letter has been taken as an (invalid, hence ignored) printer control character, not printed out; the contents of the first named memory location produce the first three characters printed (this includes one blank character for the second line), and the other two memory locations in the list are printed out in format A3, i.e., the left-most three characters in each location are printed. Thus, character strings can be input and output, and even manipulated in between; however, other programming languages (e.g., COBOL) are better suited to character manipulation than is FORTRAN.

e) There are two types of non-standard input/output. The first, *free-format I/O,* is done with "READ,*List*" and "PRINT,*List*", see page 166. The second, *FORTRAN II type I/O,* was mentioned in a footnote on page 136. The commands are "READ *m,List*" and "PRINT *m,List*", where *m* is the statement number of a FORMAT statement (i.e., the I/O is formatted) and *List* is the usual I/O list.

f) The *logical IF statement* is an alternative to the arithmetic IF of Basic FORTRAN, particularly useful for two-way branching. For example, suppose we want to print out the value of X whenever A is greater than 12.723. We can do so by means of:

```
    IF(A.GT.12.723)PRINT,X
```

where ".GT." means "greater than". The executable statement "PRINT,X" is *obeyed* if the condition "A.GT.12.723" is *true*; the statement "PRINT,X" is *ignored* if the condition is *false*, at the moment this point is reached during the execution phase.

On MIDITRAN cards, the "IF(" comes from a mark in area II, but the rest, including the keyword "PRINT", comes from area III. This is the one and only case in which the MIDITRAN compiler recognizes FORTRAN keywords in area III. When assigning line numbers, the compiler assigns *two* separate line numbers for a logical IF statement: for example, line number 12 might be assigned to "IF(A.GT.12.723)" and line number 13 to "PRINT,X". On the listing, everything is on one line and that line is numbered *012. No line number 13 appears on the listing. But if there is an execution time error, then the error message refers to line 12 for an error in connection with the condition A.GT.12.723, but to line 13 for an error in the subsequent executable statement.

The *general form of the logical IF statement* is:

```
    IF(condition) statement
```

where *statement* is any executable statement *other than* a DO statement or another logical IF statement; and where *condition* is either a simple condition or a compound condition. The *simple conditions* are: J.GT.K (J greater than K), J.GE.K (J greater than or equal to K), J.EQ.K (J equal to K), J.NE.K (J not equal to K), J.LE.K (J less than or equal to K), and J.LT.K (J less than K). In all cases, J and K must be arithmetic expressions of the *same type* (both integer or both real). If real numbers are being compared, the comparisons .GE., .EQ., .NE., and .LE. are dangerous and should be avoided, since they can be affected by round-off errors. (See page 60). *Compound conditions* contain the logical operators ".NOT.", ".AND.", or ".OR.". Let P be some simple condition (for example "A.GT.B"). Then the condition ".NOT.P" is true if P is false, and is false if P is true. (Challenge: Prove that ".NOT.A.GT.B" is equivalent to "A.LE.B"). Let P and Q be two separate simple conditions. Then the condition "P.AND.Q" is true if *both* P is true and Q is true; and is false otherwise. The condition "P.OR.Q" is true if *either* P is true *or* Q is true *or* both are true; "P.OR.Q" is false if *both* P is false and Q is false.

When executing a logical IF statement "IF(*condition*) *statement*" the computer starts by testing the *condition.* If the *condition* is *true,* the *statement* is obeyed; if the *condition* is *false,* the *statement* is ignored and execution proceeds to the immediately following line in the FORTRAN program.

Although MIDITRAN allows both simple and compound conditions, it does *not* allow bracketing, i.e., the condition "((A.GT.B.OR.C.GT.D).AND.E.GT.F)" is *not* accepted by the compiler.

7. *Extensions Involving Subprograms.*

a) A FUNCTION subprogram is allowed to alter the values of its arguments.

b) In invoking a SUBROUTINE (but *not* a FUNCTION) subprogram, the actual argument corresponding to a dummy argument of type integer may be a Hollerith constant.

c) *Arrays with variable dimension,* for example "DIMENSION OLIST(JDIM)" rather than "DIMENSION OLIST(22)", are permitted in *subprograms,* provided both the *array name and the names of the variable dimensions are dummy arguments of the subprogram, and provided that the actual arguments for the variable dimensions agree with the dimensions assigned to the actual array name in the superior program.* This feature of full FORTRAN is a great convenience when writing array manipulation subprograms, which in Basic FORTRAN have to be recompiled afresh whenever we wish to manipulate an array of some different dimension. For example, the FUNCTION INDEX(X,OLIST,N) given on pages 127-128 declares DIMENSION OLIST(22), and therefore *requires* that the superior program declares BRACK(22) (see page 129). In MIDITRAN, we can get around this by replacing the first two non-comment statements of the subprogram by:

```
FUNCTION INDEX(X,OLIST,N,JDIM)
DIMENSION OLIST(JDIM)
```

and (in the main program) replacing the actual invocation, statement 2500 on page 129, by

```
2500 KRACK=INDEX(TINC,BRACK,22,22)
```

where the first "22" sets N, the number of array elements actually used; and the second "22" sets JDIM to agree with the declared dimension for BRACK in the main program. It is possible that we might wish to use only the first 15 array elements of the array BRACK, in which case we can write: "KRACK=INDEX(TINC,BRACK,15,22)".

d) *Subprogram names as arguments.* In more advanced programming, function names may be wanted as arguments of another function. For example, to evaluate the integral of the function FUNC(X) between X=A and X=B by Simpson's rule with N intervals, we may write a function subprogram

```
FUNCTION GRAL(FUNC,A,B,N)
```

in which the first dummy argument, FUNC, is the name of a subprogram rather than the name of a memory location. No change is required within FUNCTION GRAL. To evaluate the sum of two integrals, one of the function FUNCA, the other of the function FUNCB, we must of course supply FUNCTION subprograms for FUNCA and FUNCB, again without change. The only new thing is in the main program which uses all this. Right at its beginning, an EXTERNAL declaration is required to inform the compiler that FUNCA and FUNCB are intended to be subprogram names, rather than names of memory locations. Thus:

```
C     SAMPLE MAIN PROGRAM
      EXTERNAL FUNCA,FUNCB
      . . .
      Y=GRAL(FUNCA,1.5,2.5,64) + GRAL(FUNCB,3.7,6.7,256)
      . . .
```

Subprogram names as arguments may be names of FUNCTION or SUBROUTINE subprograms, and may be used within FUNCTION or SUBROUTINE subprograms. Naturally, the actual arguments and dummy arguments must agree in type etc.

e) Last *and* least, explicit type declarations INTEGER or REAL are permitted for arguments of a subprogram, and (in a slightly different form) for the type (integer or real) of the value returned by a function subprogram. For example, suppose we wish FUNCTION RULER(IDIOT,FOOL) to have a REAL IDIOT argument, and to return an INTEGER RULER function value. We then write:

```
INTEGER FUNCTION RULER(IDIOT,FOOL)
REAL IDIOT
```

Note that the type of a parameter (dummy argument) is declared within the body of the function subprogram, but the type of the function name is declared as part of the function declaration itself. With MIDITRAN cards, *both* "INTEGER" and "FUNCTION" positions are marked in area II, the remainder in area III. The corresponding thing is done for a "REAL FUNCTION". *We advise strongly against using type declarations to override the first letter convention of FORTRAN.* The programmer can pick his own names, anyway, so why not pick them with suitable first letters? All the nonsense above can be avoided by calling the function LORD rather than RULER, and the first argument DUNCE rather than IDIOT.

INDEX

BASIC FORTRAN IV PROGRAMMING

JOHN M. BLATT
Professor of Applied Mathematics
University of New South Wales
Sydney, Australia

* A *rapid* introduction ; *no computer experience or mathematical training* needed.
* *Standard* FORTRAN used, so student can program *any* computer afterwards.
* Programming exercises on computer, *from Chapter I on.*
* Exercises in *data processing and scientific computing* ; list searching, merging.
* Answers to " drill exercises " in book, to " programming exercises " in teacher's manual.
* Easy learning in stages ; but has 3 page " Glossary of FORTRAN " for quick reference.
* Contains everything needed for practical teaching. No mimeographed handouts! No mysterious manufacturers' manuals! The *last four chapters* provide :
  ** Detailed instructions for preparing input cards, with diagrams.
  ** Structure of the input job deck, with all control cards.
  ** Explanation of the printed output produced by your computer.
  ** List of *all diagnostic messages*, all execution time error messages, etc.
  ** List of all special features of FORTRAN for your machine.
* Above material available in several " versions ", to suit your teaching needs, with your computer. Order the version fitting *your* requirements. The differences occupy only a few pages at the end. The bulk of the book, the structure of the course based on it, all the exercises, and all their solutions—all that is the *same for all versions.*

COMPUTER SYSTEMS (AUSTRALIA) PTY. LTD.
SYDNEY, AUSTRALIA